



# **MIPS32® M14K™ Processor Core Software User's Manual**

**Document Number: MD00668**

**Revision 02.03**

**April 30, 2012**

**MIPS Technologies, Inc.  
955 East Arques Avenue  
Sunnyvale, CA 94085-4521**

**Copyright © 2009, 2010 MIPS Technologies Inc. All rights reserved.**



Unpublished rights (if any) reserved under the copyright laws of the United States of America and other countries.

This document contains information that is proprietary to MIPS Technologies, Inc. ("MIPS Technologies"). Any copying, reproducing, modifying or use of this information (in whole or in part) that is not expressly permitted in writing by MIPS Technologies or an authorized third party is strictly prohibited. At a minimum, this information is protected under unfair competition and copyright laws. Violations thereof may result in criminal penalties and fines.

Any document provided in source format (i.e., in a modifiable form such as in FrameMaker or Microsoft Word format) is subject to use and distribution restrictions that are independent of and supplemental to any and all confidentiality restrictions. UNDER NO CIRCUMSTANCES MAY A DOCUMENT PROVIDED IN SOURCE FORMAT BE DISTRIBUTED TO A THIRD PARTY IN SOURCE FORMAT WITHOUT THE EXPRESS WRITTEN PERMISSION OF MIPS TECHNOLOGIES, INC.

MIPS Technologies reserves the right to change the information contained in this document to improve function, design or otherwise. MIPS Technologies does not assume any liability arising out of the application or use of this information, or of any error or omission in such information. Any warranties, whether express, statutory, implied or otherwise, including but not limited to the implied warranties of merchantability or fitness for a particular purpose, are excluded. Except as expressly provided in any written license agreement from MIPS Technologies or an authorized third party, the furnishing of this document does not give recipient any license to any intellectual property rights, including any patent rights, that cover the information in this document.

The information contained in this document shall not be exported, reexported, transferred, or released, directly or indirectly, in violation of the law of any country or international law, regulation, treaty, Executive Order, statute, amendments or supplements thereto. Should a conflict arise regarding the export, reexport, transfer, or release of the information contained in this document, the laws of the United States of America shall be the governing law.

The information contained in this document constitutes one or more of the following: commercial computer software, commercial computer software documentation or other commercial items. If the user of this information, or any related documentation of any kind, including related technical data or manuals, is an agency, department, or other entity of the United States government ("Government"), the use, duplication, reproduction, release, modification, disclosure, or transfer of this information, or any related documentation of any kind, is restricted in accordance with Federal Acquisition Regulation 12.212 for civilian agencies and Defense Federal Acquisition Regulation Supplement 227.7202 for military agencies. The use of this information by the Government is further restricted in accordance with the terms of the license agreement(s) and/or applicable contract terms and conditions covering this information from MIPS Technologies or an authorized third party.

MIPS, MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPS-3D, MIPS16e, MIPS32, MIPS64, microMIPS, MIPS-Based, MIPSsim, MIPS Logo, MIPS Technologies Logo, MIPS-VERIFIED, MIPS-VERIFIED Logo, SmartMIPS, 4K, 4Kc, 4Kp, 4KE, 4KEc, 4KEm, 4KEp, 4KEPro, 4KS, 4KSd, M4K, M14K, M14kC, 24K, 24Kc, 24Kf, 24KE, 24KEc, 24KEf, 34K, 34Kc, 34Kf, 74K, 74Kc, 74Kf, 1004K, 1004Kc, 1004Kf, R3000, R4000, R5000, R10000, "At the core of the user experience.", BusBridge, Clam, CorExtend, CoreFPGA, CoreLV, EC, FPGA View, MIPS Navigator, MIPS System Navigator, Malta, MDMX, OCI, PDtrace, Pro Series, SEAD-3, SOC-it, and YAMON are trademarks or registered trademarks of MIPS Technologies, Inc. in the United States and other countries.

# Table of Contents

<b>Chapter 1: Introduction to the MIPS32® M14K™ Processor Core .....</b>	<b>15</b>
1.1: Features .....	15
1.2: M14K™ Core Block Diagram .....	19
1.2.1: Required Logic Blocks .....	20
1.2.2: Optional Logic Blocks.....	24
<b>Chapter 2: Pipeline of the M14K™ Core .....</b>	<b>31</b>
2.1: Pipeline Stages.....	31
2.1.1: I Stage: Instruction Fetch .....	32
2.1.2: E Stage: Execution.....	33
2.1.3: M Stage: Memory Fetch.....	33
2.1.4: A Stage: Align .....	33
2.1.5: W Stage: Writeback .....	34
2.2: Multiply/Divide Operations .....	34
2.3: MDU Pipeline (High-Performance MDU) .....	34
2.3.1: 32x16 Multiply (High-Performance MDU) .....	37
2.3.2: 32x32 Multiply (High-Performance MDU) .....	37
2.3.3: Divide (High-Performance MDU) .....	38
2.4: MDU Pipeline (Area-Efficient MDU) .....	39
2.4.1: Multiply (Area-Efficient MDU).....	40
2.4.2: Multiply Accumulate (Area-Efficient MDU) .....	40
2.4.3: Divide (Area-Efficient MDU) .....	41
2.5: Branch Delay .....	41
2.6: Data Bypassing .....	42
2.6.1: Load Delay .....	43
2.6.2: Move from HI/LO and CP0 Delay.....	43
2.7: Coprocessor 2 Instructions .....	44
2.8: Interlock Handling.....	45
2.9: Slip Conditions.....	46
2.10: Instruction Interlocks.....	46
2.11: Hazards .....	47
2.11.1: Types of Hazards .....	48
2.11.2: Instruction Listing .....	49
2.11.3: Eliminating Hazards .....	49
<b>Chapter 3: Memory Management of the M14K™ Core .....</b>	<b>51</b>
3.1: Introduction.....	51
3.2: Modes of Operation .....	51
3.2.1: Virtual Memory Segments.....	52
3.2.2: User Mode.....	54
3.2.3: Kernel Mode.....	55
3.2.4: Debug Mode.....	57
3.3: Fixed Mapping MMU .....	59
3.4: System Control Coprocessor.....	61
<b>Chapter 4: Exceptions and Interrupts in the M14K™ Core.....</b>	<b>63</b>

4.1: Exception Conditions .....	63
4.2: Exception Priority .....	64
4.3: Interrupts .....	65
4.3.1: Interrupt Modes .....	65
4.3.2: Generation of Exception Vector Offsets for Vectored Interrupts .....	74
4.3.3: MCU ASE Enhancement for Interrupt Handling .....	75
4.4: GPR Shadow Registers .....	76
4.5: Exception Vector Locations .....	77
4.6: General Exception Processing .....	79
4.7: Debug Exception Processing .....	81
4.8: Exception Descriptions .....	82
4.8.1: Reset/SoftReset Exception .....	82
4.8.2: Debug Single Step Exception .....	83
4.8.3: Debug Interrupt Exception .....	84
4.8.4: Non-Maskable Interrupt (NMI) Exception .....	84
4.8.5: Interrupt Exception .....	85
4.8.6: Debug Instruction Break Exception .....	85
4.8.7: Address Error Exception — Instruction Fetch/Data Access .....	85
4.8.8: SRAM Parity Error Exception .....	86
4.8.9: Bus Error Exception — Instruction Fetch or Data Access .....	86
4.8.10: Protection Exception .....	87
4.8.11: Debug Software Breakpoint Exception .....	87
4.8.12: Execution Exception — System Call .....	87
4.8.13: Execution Exception — Breakpoint .....	88
4.8.14: Execution Exception — Reserved Instruction .....	88
4.8.15: Execution Exception — Coprocessor Unusable .....	88
4.8.16: Execution Exception — CorExtend Unusable .....	89
4.8.17: Execution Exception — Coprocessor 2 Exception .....	89
4.8.18: Execution Exception — Implementation-Specific 1 Exception .....	89
4.8.19: Execution Exception — Integer Overflow .....	90
4.8.20: Execution Exception — Trap .....	90
4.8.21: Debug Data Break Exception .....	90
4.8.22: Complex Break Exception .....	91
4.9: Exception Handling and Servicing Flowcharts .....	91

## **Chapter 5: CP0 Registers of the M14K™ Core ..... 95**

5.1: CP0 Register Summary .....	95
5.2: CP0 Register Descriptions .....	97
5.2.1: UserLocal Register (CP0 Register 4, Select 2) .....	97
5.2.2: HWREna Register (CP0 Register 7, Select 0) .....	98
5.2.3: BadVAddr Register (CP0 Register 8, Select 0) .....	99
5.2.4: Count Register (CP0 Register 9, Select 0) .....	99
5.2.5: Compare Register (CP0 Register 11, Select 0) .....	100
5.2.6: Status Register (CP0 Register 12, Select 0) .....	100
5.2.7: IntCtl Register (CP0 Register 12, Select 1) .....	104
5.2.8: SRSCtl Register (CP0 Register 12, Select 2) .....	108
5.2.9: SRSMap Register (CP0 Register 12, Select 3) .....	111
5.2.10: View_IPL Register (CP0 Register 12, Select 4) .....	112
5.2.11: SRSMap2 Register (CP0 Register 12, Select 5) .....	112
5.2.12: Cause Register (CP0 Register 13, Select 0) .....	113
5.2.13: View_RIPL Register (CP0 Register 13, Select 4) .....	118
5.2.14: NestedExc (CP0 Register 13, Select 5) .....	118
5.2.15: Exception Program Counter (CP0 Register 14, Select 0) .....	119

5.2.16: NestedEPC (CP0 Register 14, Select 2).....	120
5.2.17: Processor Identification (CP0 Register 15, Select 0) .....	121
5.2.18: EBase Register (CP0 Register 15, Select 1) .....	122
5.2.19: CDMMBase Register (CP0 Register 15, Select 2).....	123
5.2.20: Config Register (CP0 Register 16, Select 0).....	124
5.2.21: Config1 Register (CP0 Register 16, Select 1).....	126
5.2.22: Config2 Register (CP0 Register 16, Select 2).....	127
5.2.23: Config3 Register (CP0 Register 16, Select 3).....	128
5.2.24: Configuration Register 4 (CP0 Register 16, Select 4).....	131
5.2.25: Config5 Register (CP0 Register 16, Select 5).....	132
5.2.26: Config7 Register (CP0 Register 16, Select 7).....	132
5.2.27: Debug Register (CP0 Register 23, Select 0) .....	133
5.2.28: Trace Control Register (CP0 Register 23, Select 1) .....	137
5.2.29: Trace Control2 Register (CP0 Register 23, Select 2) .....	139
5.2.30: User Trace Data1 Register (CP0 Register 23, Select 3)/User Trace Data2 Register (CP0 Register 24, Select 3) .....	141
5.2.31: TraceBPC Register (CP0 Register 23, Select 4) .....	142
5.2.32: Debug2 Register (CP0 Register 23, Select 6) .....	143
5.2.33: Debug Exception Program Counter Register (CP0 Register 24, Select 0) .....	144
5.2.34: Performance Counter Register (CP0 Register 25, select 0-3) .....	145
5.2.35: ErrCtl Register (CP0 Register 26, Select 0).....	150
5.2.36: CacheErr Register (CP0 Register 27, Select 0).....	150
5.2.37: ErrorEPC (CP0 Register 30, Select 0).....	151
5.2.38: DeSave Register (CP0 Register 31, Select 0) .....	152
<b>Chapter 6: Hardware and Software Initialization of the M14K™ Core .....</b>	<b>153</b>
6.1: Hardware-Initialized Processor State .....	153
6.1.1: Coprocessor 0 State .....	153
6.1.2: Bus State Machines .....	154
6.1.3: Static Configuration Inputs .....	154
6.1.4: Fetch Address.....	154
6.2: Software Initialized Processor State.....	154
6.2.1: Register File .....	154
6.2.2: Coprocessor 0 State .....	154
<b>Chapter 7: Power Management of the M14K™ Core .....</b>	<b>157</b>
7.1: Register-Controlled Power Management .....	157
7.2: Instruction-Controlled Power Management .....	158
<b>Chapter 8: EJTAG Debug Support in the M14K™ Core .....</b>	<b>159</b>
8.1: Debug Control Register .....	159
8.2: Hardware Breakpoints .....	163
8.2.1: Data Breakpoints.....	164
8.2.2: Complex Breakpoints.....	164
8.2.3: Conditions for Matching Breakpoints .....	165
8.2.4: Debug Exceptions from Breakpoints.....	168
8.2.5: Breakpoint Used as Triggerpoint.....	169
8.2.6: Instruction Breakpoint Registers .....	170
8.2.7: Data Breakpoint Registers .....	175
8.2.8: Complex Breakpoint Registers.....	182
8.3: Complex Breakpoint Usage.....	186
8.3.1: Checking for Presence of Complex Break Support.....	186

8.3.2: General Complex Break Behavior.....	187
8.3.3: Usage of Pass Counters .....	188
8.3.4: Usage of Tuple Breakpoints.....	188
8.3.5: Usage of Priming Conditions.....	188
8.3.6: Usage of Data Qualified Breakpoints .....	189
8.3.7: Usage of Stopwatch Timers .....	189
8.4: Test Access Port (TAP) .....	190
8.4.1: EJTAG Internal and External Interfaces.....	190
8.4.2: Test Access Port Operation .....	191
8.4.3: Test Access Port (TAP) Instructions .....	194
8.5: EJTAG TAP Registers.....	197
8.5.1: Instruction Register .....	197
8.5.2: Data Registers Overview .....	198
8.5.3: Processor Access Address Register.....	206
8.5.4: Fastdata Register (TAP Instruction FASTDATA) .....	207
8.6: TAP Processor Accesses .....	208
8.6.1: Fetch/Load and Store from/to EJTAG Probe Through dmseg .....	209
8.7: SecureDebug.....	210
8.7.1: Disabling EJTAG debugging .....	210
8.7.2: EJTAG Features Unmodified by SecureDebug .....	211
8.8: iFlowtrace™ Mechanism .....	211
8.8.1: A Simple Instruction-Only Tracing Scheme .....	212
8.8.2: Special Trace Modes .....	213
8.8.3: ITCB Overview .....	216
8.8.4: ITCB iFlowTrace Interface .....	217
8.8.5: TCB Storage Representation .....	217
8.8.6: ITCB Register Interface for Software Configurability .....	218
8.8.7: ITCB iFlowtrace Off-Chip Interface .....	222
8.8.8: Breakpoint-Based Enabling of Tracing.....	222
8.9: PC/Data Address Sampling.....	223
8.9.1: PC Sampling in Wait State.....	224
8.9.2: Data Address Sampling .....	224
8.10: Fast Debug Channel.....	224
8.10.1: Common Device Memory Map.....	225
8.10.2: Fast Debug Channel Interrupt.....	225
8.10.3: M14K™M14K FDC Buffers .....	225
8.10.4: Sleep mode .....	227
8.10.5: FDC TAP Register .....	227
8.10.6: Fast Debug Channel Registers .....	228
8.11: cJTAG Interface.....	232

<b>Chapter 9: Instruction Set Overview .....</b>	<b>233</b>
9.1: CPU Instruction Formats .....	233
9.2: Load and Store Instructions.....	234
9.2.1: Scheduling a Load Delay Slot.....	234
9.2.2: Defining Access Types.....	234
9.3: Computational Instructions .....	235
9.3.1: Cycle Timing for Multiply and Divide Instructions.....	236
9.4: Jump and Branch Instructions .....	236
9.4.1: Overview of Jump Instructions .....	236
9.4.2: Overview of Branch Instructions .....	236
9.5: Control Instructions.....	236
9.6: Coprocessor Instructions.....	236

9.7: Enhancements to the MIPS Architecture.....	236
9.7.1: CLO - Count Leading Ones.....	237
9.7.2: CLZ - Count Leading Zeros.....	237
9.7.3: MADD - Multiply and Add Word .....	237
9.7.4: MADDU - Multiply and Add Unsigned Word .....	237
9.7.5: MSUB - Multiply and Subtract Word .....	237
9.7.6: MSUBU - Multiply and Subtract Unsigned Word.....	238
9.7.7: MUL - Multiply Word.....	238
9.7.8: SSNOP- Superscalar Inhibit NOP.....	238
9.8: MCU ASE Instructions.....	238
9.8.1: ACLR.....	238
9.8.2: ASET .....	238
9.8.3: IRET .....	238
<b>Chapter 10: M14K™ Processor Core Instructions .....</b>	<b>239</b>
10.1: Understanding the Instruction Descriptions.....	239
10.2: M14K™ Opcode Map .....	239
10.3: MIPS32® Instruction Set for the M14K™ core .....	242
ACLR.....	249
ASET .....	253
IRET .....	257
LL .....	265
PREF.....	267
SC .....	269
SYNC .....	272
WAIT .....	273
<b>Chapter 11: microMIPS™ Instruction Set Architecture .....</b>	<b>275</b>
11.1: Overview.....	275
11.1.1: MIPSr3™ Architecture.....	275
11.1.2: Default ISA Mode .....	276
11.1.3: Software Detection.....	276
11.1.4: Compliance and Subsetting .....	276
11.1.5: Mode Switch.....	276
11.1.6: Branch and Jump Offsets.....	277
11.1.7: Coprocessor Unusable Behavior .....	277
11.2: Instruction Formats.....	277
11.2.1: Instruction Stream Organization and Endianness.....	281
11.3: microMIPS Re-encoded Instructions .....	281
11.3.1: 16-Bit Category .....	282
11.3.2: 16-bit Instruction Register Set.....	287
11.3.3: 32-Bit Category .....	289
<b>Appendix A: References .....</b>	<b>291</b>
<b>Appendix B: Revision History .....</b>	<b>293</b>

# List of Figures

Figure 1: M14K™ Processor Core Block Diagram .....	20
Figure 1.1: Address Translation During a SRAM Access .....	23
Figure 1.2: Reference Design Block Diagram.....	25
Figure 1.3: FDC Overview.....	27
Figure 1.4: cJTAG Support .....	28
Figure 2.1: M14K™ Core Pipeline Stages (with high-performance MDU) .....	32
Figure 2.2: M14K™ Core Pipeline Stages (with area-efficient MDU) .....	32
Figure 2.3: MDU Pipeline Behavior During Multiply Operations .....	36
Figure 2.4: MDU Pipeline Flow During a 32x16 Multiply Operation .....	37
Figure 2.5: MDU Pipeline Flow During a 32x32 Multiply Operation .....	38
Figure 2.6: High-Performance MDU Pipeline Flow During a 8-bit Divide (DIV) Operation .....	38
Figure 2.7: High-Performance MDU Pipeline Flow During a 16-bit Divide (DIV) Operation .....	38
Figure 2.8: High-Performance MDU Pipeline Flow During a 24-bit Divide (DIV) Operation .....	39
Figure 2.9: High-Performance MDU Pipeline Flow During a 32-bit Divide (DIV) Operation .....	39
Figure 2.10: M14K™ Area-Efficient MDU Pipeline Flow During a Multiply Operation .....	40
Figure 2.11: M14K Area-Efficient MDU Pipeline Flow During a Multiply Accumulate Operation .....	40
Figure 2.12: M14K™ Area-Efficient MDU Pipeline Flow During a Divide (DIV) Operation .....	41
Figure 2.13: IU Pipeline Branch Delay .....	42
Figure 2.14: IU Pipeline Data Bypass .....	42
Figure 2.15: IU Pipeline M to E bypass .....	43
Figure 2.16: IU Pipeline A to E Data bypass .....	43
Figure 2.17: IU Pipeline Slip after a MFHI .....	44
Figure 2.18: Coprocessor 2 Interface Transactions .....	45
Figure 2.19: Instruction Cache Miss Slip .....	46
Figure 3.1: Address Translation During SRAM Access .....	51
Figure 3.2: M14K™ processor core Virtual Memory Map .....	53
Figure 3.3: User Mode Virtual Address Space .....	54
Figure 3.4: Kernel Mode Virtual Address Space .....	56
Figure 3.5: Debug Mode Virtual Address Space .....	58
Figure 3.6: FM Memory Map (ERL=0) in the M14K™ Processor Core .....	60
Figure 3.7: FM Memory Map (ERL=1) in the M14K™ Processor Core .....	61
Figure 4.1: Interrupt Generation for Vectored Interrupt Mode .....	70
Figure 4.2: Interrupt Generation for External Interrupt Controller Interrupt Mode .....	73
Figure 4.3: General Exception Handler (HW) .....	92
Figure 4.4: General Exception Servicing Guidelines (SW) .....	93
Figure 4.5: Reset, Soft Reset and NMI Exception Handling and Servicing Guidelines .....	94
Figure 5.1: UserLocal Register Format .....	97
Figure 5.2: HWREna Register Format .....	98
Figure 5.3: BadVAddr Register Format .....	99
Figure 5.4: Count Register Format .....	99
Figure 5.5: Compare Register Format .....	100
Figure 5.6: Status Register Format.....	101
Figure 5.7: IntCtl Register Format.....	105
Figure 5.8: SRSCtl Register Format .....	109
Figure 5.9: SRSMap Register Format.....	111
Figure 5-10: View_IPL Register Format.....	112
Figure 5-11: SRSMap Register Format.....	113



Figure 5.12: Cause Register Format.....	113
Figure 5.13: View_RIPL Register Format .....	118
Figure 5.14: NestedExc Register Format.....	119
Figure 5.15: EPC Register Format .....	120
Figure 5.16: NestedEPC Register Format .....	121
Figure 5.17: PRId Register Format .....	121
Figure 5.18: EBase Register Format.....	123
Figure 5.19: CDMMBase Register Format.....	123
Figure 5.20: Config Register Format — Select 0 .....	124
Figure 5.21: Config Register Field Descriptions.....	124
Figure 5.22: Config1 Register Format — Select 1 .....	126
Figure 5.23: Config2 Register Format — Select 2 .....	127
Figure 5.24: Config3 Register Format.....	128
Figure 5.25: Config4 Register Format.....	131
Figure 5.26: Config5 Register Format.....	132
Figure 5.27: Config7 Register Format .....	132
Figure 5.28: Debug Register Format .....	133
Figure 5.29: TraceControl Register Format .....	137
Figure 5.30: TraceControl2 Register Format .....	139
Figure 5.31: User Trace Data1/User Trace Data2 Register Format .....	141
Figure 5.32: Trace BPC Register Format .....	142
Figure 5.33: Debug2 Register Format .....	143
Figure 5.34: DEPC Register Format .....	144
Figure 5.35: Performance Counter Control Register .....	145
Figure 5.36: Performance Counter Count Register .....	149
Figure 5.37: ErrCtl Register Format .....	150
Figure 5.38: CacheErr Register (Primary Caches) .....	150
Figure 5.39: ErrorEPC Register Format .....	152
Figure 5.40: DeSave Register Format .....	152
Figure 8.1: DCR Register Format .....	160
Figure 8.2: IBS Register Format .....	170
Figure 8.3: IBAn Register Format .....	171
Figure 8.4: IBMn Register Format .....	171
Figure 8.5: IBASIDn Register Format .....	172
Figure 8.6: IBCn Register Format .....	172
Figure 8.7: IBCCn Register Format .....	174
Figure 8.8: IBPCn Register Format .....	175
Figure 8.9: DBS Register Format .....	176
Figure 8.10: DBAn Register Format .....	176
Figure 8.11: DBMn Register Format .....	177
Figure 8.12: DBASIDn Register Format .....	177
Figure 8.13: DBCn Register Format .....	178
Figure 8.14: DBVn Register Format .....	179
Figure 8.15: DBCCn Register Format .....	180
Figure 8.16: DBPCn Register Format .....	181
Figure 8.17: DVM Register Format .....	181
Figure 8.18: CBTC Register Format .....	182
Figure 8.19: PrCndA Register Format .....	183
Figure 8.20: STCtl Register Format .....	185
Figure 8.21: STCnt Register Format .....	186
Figure 8.22: TAP Controller State Diagram .....	192
Figure 8.23: Concatenation of the EJTAG Address, Data and Control Registers .....	196
Figure 8.24: TDI to TDO Path When in Shift-DR State and FASTDATA Instruction is Selected .....	197

Figure 8.25: Device Identification Register Format .....	198
Figure 8.26: Implementation Register Format .....	199
Figure 8.27: EJTAG Control Register Format .....	200
Figure 8.28: Endian Formats for the PAD Register .....	207
Figure 8.29: Fastdata Register Format .....	207
Figure 8.30: Trace Logic Overview .....	216
Figure 8.31: Control/Status Register.....	218
Figure 8.32: ITCBTW Register Format .....	220
Figure 8.33: ITCBRDP Register Format .....	221
Figure 8.34: ITCBWRP Register Format.....	221
Figure 8.35: PCSAMPLE TAP Register Format (MIPS32) .....	223
Figure 8.36: Fast Debug Channel Buffer Organization .....	226
Figure 8.37: FDC TAP Register Format.....	227
Figure 8.38: FDC Access Control and Status Register.....	228
Figure 8.39: FDC Configuration Register .....	229
Figure 8.40: FDC Status Register .....	230
Figure 8.41: FDC Receive Register .....	231
Figure 8.42: FDC Transmit Register .....	231
Figure 8.43: cJTAG Interface .....	232
Figure 9.1: Instruction Formats .....	234
Figure 11.1: 16-Bit Instruction Formats.....	279
Figure 11.2: 32-Bit Instruction Formats.....	280
Figure 11.3: Immediate Fields within 32-Bit Instructions.....	280

# List of Tables

Table 2.1: MDU Instruction Latencies (High-Performance MDU) .....	35
Table 2.2: MDU Instruction Repeat Rates (High-Performance MDU).....	36
Table 2.3: M14K™ Core Instruction Latencies (Area-Efficient MDU) .....	39
Table 2.4: Pipeline Interlocks .....	45
Table 2.5: Instruction Interlocks .....	47
Table 2.6: Execution Hazards .....	48
Table 2.7: Instruction Hazards .....	48
Table 2.8: Hazard Instruction Listing .....	49
Table 3.1: User Mode Segments .....	54
Table 3.2: Kernel Mode Segments .....	56
Table 3.3: Physical Address and Cache Attributes for dseg, dmseg, and drseg Address Spaces .....	58
Table 3.4: CPU Access to drseg Address Range .....	58
Table 3.5: CPU Access to dmseg Address Range .....	59
Table 3.6: Cacheability of Segments with Block Address Translation .....	59
Table 4.1: Priority of Exceptions .....	64
Table 4.2: Interrupt Modes .....	66
Table 4.3: Relative Interrupt Priority for Vectored Interrupt Mode.....	69
Table 4.4: Exception Vector Offsets for Vectored Interrupts.....	74
Table 4.5: Exception Vector Base Addresses.....	78
Table 4.6: Exception Vector Offsets .....	78
Table 4.7: Exception Vectors .....	78
Table 4.8: Value Stored in EPC, ErrorEPC, or DEPC on an Exception.....	79
Table 4.9: Debug Exception Vector Addresses .....	82
Table 4.10: Register States an Interrupt Exception .....	85
Table 4.11: CP0 Register States on an Address Exception Error.....	86
Table 4.12: CP0 Register States on a SRAM Parity Error Exception .....	86
Table 4.13: Register States on a Coprocessor Unusable Exception .....	89
Table 5.1: CP0 Registers .....	95
Table 5.2: CP0 Register Field Types .....	97
Table 5.4: HWREna Register Field Descriptions .....	98
Table 5.3: UserLocal Register Field Descriptions .....	98
Table 5.5: BadVAddr Register Field Description.....	99
Table 5.6: Count Register Field Description .....	99
Table 5.7: Compare Register Field Description .....	100
Table 5.8: Status Register Field Descriptions .....	101
Table 5.9: IntCtl Register Field Descriptions.....	105
Table 5.10: SRSCtl Register Field Descriptions .....	109
Table 5.11: Sources for new SRSCtl <sub>CSS</sub> on an Exception or Interrupt .....	111
Table 5.12: SRSMap Register Field Descriptions.....	111
Table 5.13: View_IPL Register Field Descriptions .....	112
Table 5.14: SRSMap Register Field Descriptions.....	113
Table 5.15: Cause Register Field Descriptions.....	113
Table 5.16: Cause Register ExcCode Field.....	117
Table 5.17: View_RIPL Register Field Descriptions .....	118
Table 5.18: NestedExc Register Field Descriptions.....	119
Table 5.19: EPC Register Field Description.....	120
Table 5.20: NestedEPC Register Field Descriptions .....	121

Table 5.21: PRId Register Field Descriptions .....	121
Table 5.22: EBase Register Field Descriptions.....	123
Table 5.23: CDMMBase Register Field Descriptions.....	123
Table 5.24: Cache Coherency Attributes .....	126
Table 5.25: Config1 Register Field Descriptions — Select 1 .....	126
Table 5.26: Config2 Register Field Descriptions — Select 1 .....	127
Table 5.27: Config3 Register Field Descriptions.....	128
Table 5.28: Config4 Register Field Descriptions.....	131
Table 5.29: Config5 Register Field Descriptions.....	132
Table 5.30: Config7 Register Field Descriptions.....	133
Table 5.31: Debug Register Field Descriptions.....	134
Table 5.32: TraceControl Register Field Descriptions .....	137
Table 5.33: TraceControl2 Register Field Descriptions .....	139
Table 5.34: UserTraceData1/UserTraceData2 Register Field Descriptions .....	141
Table 5.35: TraceBPC Register Field Descriptions.....	142
Table 5.36: Debug2 Register Field Descriptions.....	143
Table 5.37: DEPC Register Formats.....	144
Table 5.38: Performance Counter Register Selects.....	145
Table 5.39: Performance Counter Control Register Field Descriptions .....	145
Table 5.40: Event Descriptions .....	148
Table 5.41: Performance Counter Count Register Field Descriptions .....	149
Table 5.42: Errctl Register Field Descriptions.....	150
Table 5.43: CacheErr Register Field Descriptions (Primary Caches) .....	150
Table 5.44: ErrorEPC Register Field Description.....	152
Table 5.45: DeSave Register Field Description .....	152
Table 8.1: DCR Register Field Descriptions .....	160
Table 8.2: Addresses for Instruction Breakpoint Registers .....	170
Table 8.3: IBS Register Field Descriptions .....	170
Table 8.4: IBAn Register Field Descriptions .....	171
Table 8.6: IBASIDn Register Field Descriptions .....	172
Table 8.7: IBCn Register Field Descriptions .....	172
Table 8.5: IBMn Register Field Descriptions.....	172
Table 8.8: IBCCn Register Field Descriptions.....	174
Table 8.9: IBPCn Register Field Descriptions.....	175
Table 8.10: Addresses for Data Breakpoint Registers .....	175
Table 8.11: DBS Register Field Descriptions.....	176
Table 8.12: DBAn Register Field Descriptions.....	176
Table 8.13: DBMn Register Field Descriptions .....	177
Table 8.14: DBASIDn Register Field Descriptions.....	177
Table 8.15: DBCn Register Field Descriptions.....	178
Table 8.16: DBVn Register Field Descriptions.....	179
Table 8.17: DBCCn Register Field Descriptions .....	180
Table 8.18: DBPCn Register Field Descriptions .....	181
Table 8.19: DVM Register Field Descriptions .....	181
Table 8.20: Addresses for Complex Breakpoint Registers .....	182
Table 8.21: CBTC Register Field Descriptions .....	182
Table 8.23: Priming Conditions and Register Values for 6I/2D Configuration .....	184
Table 8.24: Priming Conditions and Register Values for 8I/4D Configuration .....	184
Table 8.22: PrCndA Register Field Descriptions.....	184
Table 8.25: STCtl Register Field Descriptions .....	185
Table 8.26: STCtl Register Field Descriptions .....	186
Table 8.27: EJTAG Interface Pins .....	190
Table 8.28: Implemented EJTAG Instructions .....	195

Table 8.30: Implementation Register Descriptions .....	199
Table 8.29: Device Identification Register.....	199
Table 8.31: EJTAG Control Register Descriptions.....	201
Table 8.32: Fastdata Register Field Description .....	207
Table 8.33: Operation of the FASTDATA access .....	208
Table 8.34: EJ_DisableProbeDebug Signal Overview.....	211
Table 8.35: Data Bus Encoding .....	217
Table 8.36: Tag Bit Encoding.....	218
Table 8.37: Control/Status Register Field Descriptions .....	219
Table 8.38: ITCBTW Register Field Descriptions .....	220
Table 8.39: ITCBRDP Register Field Descriptions .....	221
Table 8.40: ITCBWRP Register Field Descriptions.....	221
Table 8.41: drseg Registers that Enable/Disable Trace from Breakpoint-Based Triggers.....	222
Table 8.42: FDC TAP Register Field Descriptions.....	227
Table 8.43: FDC Register Mapping.....	228
Table 8.44: FDC Access Control and Status Register Field Descriptions .....	228
Table 8.45: FDC Configuration Register Field Descriptions .....	229
Table 8.46: FDC Status Register Field Descriptions.....	230
Table 8.47: FDC Receive Register Field Descriptions.....	231
Table 8.49: FDTXn Address Decode .....	232
Table 8.48: FDC Transmit Register Field Descriptions.....	232
Table 9.1: Byte Access Within a Word.....	235
Table 10.1: Encoding of the <i>Opc</i> Field.....	240
Table 10.2: <i>Special</i> Opcode encoding of Function Field.....	240
Table 10.3: <i>Special2</i> Opcode Encoding of Function Field .....	240
Table 10.4: <i>Special3</i> Opcode Encoding of Function Field .....	241
Table 10.5: <i>RegImm</i> Encoding of <i>rt</i> Field.....	241
Table 10.6: <i>COP2</i> Encoding of <i>rs</i> Field .....	241
Table 10.7: <i>COP2</i> Encoding of <i>rt</i> Field When <i>rs=BC2</i> .....	241
Table 10.8: <i>COP0</i> Encoding of <i>rs</i> Field .....	242
Table 10.9: <i>COP0</i> Encoding of Function Field When <i>rs=CO</i> .....	242
Table 10.10: Instruction Set.....	242
Format:: Values of <i>hint</i> Field for PREF Instruction.....	267
Table 11.1: 16-Bit Re-encoding of Frequent MIPS32 Instructions.....	283
Table 11.2: 16-Bit Re-encoding of Frequent MIPS32 Instruction Sequences.....	284
Table 11.3: Instruction-Specific Register Specifiers and Immediate Field Values .....	286
Table 11.4: 16-Bit Instruction General-Purpose Registers - \$2-\$7, \$16, \$17 .....	287
Table 11.5: SB16, SH16, SW16 Source Registers - \$0, \$2-\$7, \$17.....	288
Table 11.6: 16-Bit Instruction Implicit General-Purpose Registers .....	288
Table 11.7: 16-Bit Instruction Special-Purpose Registers.....	289
Table 11.8: 32-bit Instructions introduced within microMIPS .....	289



# Introduction to the MIPS32® M14K™ Processor Core

The MIPS32® M14K™ core from MIPS Technologies is a high-performance, low-power, 32-bit MIPS RISC processor core intended for custom system-on-silicon applications. The core is designed for semiconductor manufacturing companies, ASIC developers, and system OEMs who want to rapidly integrate their own custom logic and peripherals with a high-performance RISC processor. An M14K core is fully synthesizable to allow maximum flexibility; it is highly portable across processes and can easily be integrated into full system-on-silicon designs. This allows developers to focus their attention on end-user specific characteristics of their product.

The M14K core is especially well-suited for microcontrollers and applications that have real-time requirements with a high level of performance efficiency and security requirements.

The M14K implements the MIPSr3™ Architecture in a 5-stage pipeline. It includes support for the MIPS32 Release 3 Architecture and microMIPS™, an Instruction Set Architecture with optimized MIPS32 16-bit and 32-bit instructions, that provides a significant reduction in code size while maintaining performance equivalent to MIPS32. The M14K is a successor to the M4K®, designed from the same microarchitecture, including the Microcontroller Application-Specific Extension (MCU™ ASE), enhanced interrupt handling, lower interrupt latency, a memory protection unit, a reference design of an optimized interface for flash memory, and native AMBA®-3 AHB-Lite Bus Interface Unit (BIU), and additional power saving, security, debug, and profiling features.

The M14K core is cacheless; in lieu of caches, it includes a simple interface to SRAM-style devices. This interface may be configured for independent instruction and data devices or combined into a unified interface. The SRAM interface allows deterministic latency to memory, while still maintaining high performance.

The MMU consists of a simple Fixed Mapping Translation (FMT) mechanism, for applications that do not require the full capabilities of a Translation Lookaside Buffer- (TLB-) based MMU available on other MIPS cores.

The core includes one of two different Multiply/Divide Unit (MDU) implementations, selectable at build-time, allowing the user to trade off performance and area for integer multiply and divide operations. The high-performance MDU option implements single-cycle multiply and multiply-accumulate (MAC) instructions, which enable DSP algorithms to be performed efficiently. It allows 32-bit x 16-bit MAC instructions to be issued every cycle, while a 32-bit x 32-bit MAC instruction can be issued every other cycle. The area-efficient MDU option handles multiplies with a one-bit-per-clock iterative algorithm.

The basic Enhanced JTAG (EJTAG) features provide CPU run control with stop, single-stepping and re-start, and with software breakpoints using the SDBBP instruction. Additional EJTAG features such as instruction and data virtual address hardware breakpoints, complex hardware breakpoints, connection to an external EJTAG probe through the Test Access Port (TAP), and PC/Data tracing, may be included as an option.

## 1.1 Features

- 5-stage pipeline
- 32-bit Address and Data Paths

- MIPS32-Compatible Instruction Set
  - Multiply-Accumulate and Multiply-Subtract Instructions (MADD, MADDDU, MSUB, MSUBU)
  - Targeted Multiply Instruction (MUL)
  - Zero/One Detect Instructions (CLZ, CLO)
  - Wait Instruction (WAIT)
  - Conditional Move Instructions (MOVZ, MOVN)
- MIPS32 Enhanced Architecture (Release 2) Features
  - Vectored interrupts and support for external interrupt controller
  - Programmable exception vector base
  - Atomic interrupt enable/disable
  - GPR shadow registers (one, three, seven, or fifteen additional shadows can be optionally added to minimize latency for interrupt handlers)
  - Bit field manipulation instructions
- microMIPS-Compatible Instruction Set
  - microMIPS ISA is a build-time configurable and run-time convertible ISA to improve code size density over MIPS32, while maintaining MIPS32 performance.
  - Combining both 16-bit and 32-bit opcodes, microMIPS supports all MIPS32 instructions (except branch-likely instructions) with new optimized encoding. Frequently used MIPS32 instructions are available as 16-bit instructions.
  - Added fifteen new 32-bit instructions and thirty-nine 16-bit instructions corresponding to commonly used MIPS32 instructions.
  - Stack pointer implicit in instruction.
  - MIPS32 assembly and ABI compatible.
  - Supports ASEs and User-defined Instructions (UDIs).
- MCU™ ASE
  - Increases the number of interrupt hardware inputs from 6 to 8 for Vectored Interrupt (VI) mode, and from 63 to 255 for External Interrupt Controller (EIC) mode.
  - Separate priority and vector generation. 16-bit vector address is provided.
  - Hardware assist, combined with the use of Shadow Register Sets to reduce interrupt latency during the prologue and epilogue of an interrupt.



- An interrupt return with automated interrupt epilogue handling instruction (IRET) improves interrupt latency.
- Supports optional interrupt chaining.
- Two memory-to-memory atomic read-modify-write instructions (ASET and ACLR) eases commonly used semaphore manipulation in microcontroller applications. Interrupts are automatically disabled during the operation to maintain coherency.
- Memory Management Unit
  - Simple Fixed Mapping Translation (FMT) mechanism
- Memory Protection Unit (MPU)
  - Optional feature that improves system security by restricting access, execution, and trace capabilities from untrusted code in predefined memory regions.
- Simple SRAM-Style Interface
  - Cacheless operation enables deterministic response and reduces die-size
  - 32-bit address and data; input byte-enables enable simple connection to narrower devices
  - Single or multi-cycle latencies
  - Configuration option for dual or unified instruction/data interfaces
  - Redirection mechanism on dual I/D interfaces permits D-side references to be handled by I-side
  - Transactions can be aborted
- Reference Design
  - A typical SRAM reference design is provided.
  - An AHB-Lite BIU reference design is provided between the SRAM interface and AHB-Lite Bus.
  - An optimized interface for slow memory (Flash) access using prefetch buffer scheme is provided.
- Parity Support
  - The I-SRAM and D-SRAM support optional parity detection.
- CorExtend® User-Defined Instruction Set Extensions
  - Allows user to define and add instructions to the core at build time
  - Maintains full MIPS32 compatibility
  - Supported by industry-standard development tools
  - Single- or multi-cycle instructions

- Multi-Core Support
  - External lock indication enables multi-processor semaphores based on LL/SC instructions
  - External sync indication allows memory ordering
  - Debug support includes cross-core triggers
- Multiply/Divide Unit (high-performance configuration)
  - Maximum issue rate of one 32x16 multiply per clock via on-chip 32x16 hardware multiplier array.
  - Maximum issue rate of one 32x32 multiply every other clock
  - Early-in iterative divide. Minimum 11 and maximum 34 clock latency (dividend (*rs*) sign extension-dependent)
- Multiply/Divide Unit (area-efficient configuration)
  - 32 clock latency on multiply
  - 34 clock latency on multiply-accumulate
  - 33-35 clock latency on divide (sign-dependent)
- Full featured Coprocessor 2 Interface
  - 32-bit interface to an external coprocessor
- Power Control
  - Minimum frequency: 0 MHz
  - Power-down mode (triggered by WAIT instruction)
  - Support for software-controlled clock divider
  - Support for extensive use of local gated clocks
- EJTAG Debug/Profiling and iFlowtrace™ Mechanism
  - CPU control with start, stop, and single-stepping
  - Virtual instruction and data address/value breakpoints
  - Hardware breakpoint supports both address match and address range triggering.
  - Optional simple hardware breakpoints on virtual addresses; 8I/4D, 6I/2D, 4I/2D, 2I/1D breakpoints, or no breakpoints
  - Optional complex hardware breakpoints with 8I/4D, 6I/2D simple breakpoints
  - TAP controller is chainable for multi-CPU debug

- Support EJTAG (IEEE 1149.1) and compatible with cJTAG 2-wire (IEEE 1149.7) extension protocol
- Cross-CPU breakpoint support
- iFlowtrace support for real-time instruction PC and special events
- PC and/or load/store address sampling for profiling
- Performance Counters
- Support for Fast Debug Channel (FDC)
- SecureDebug
  - An optional feature that disables access via EJTAG in an untrusted environment.
- Testability
  - Full scan design achieves test coverage in excess of 99% (dependent on library and configuration options).

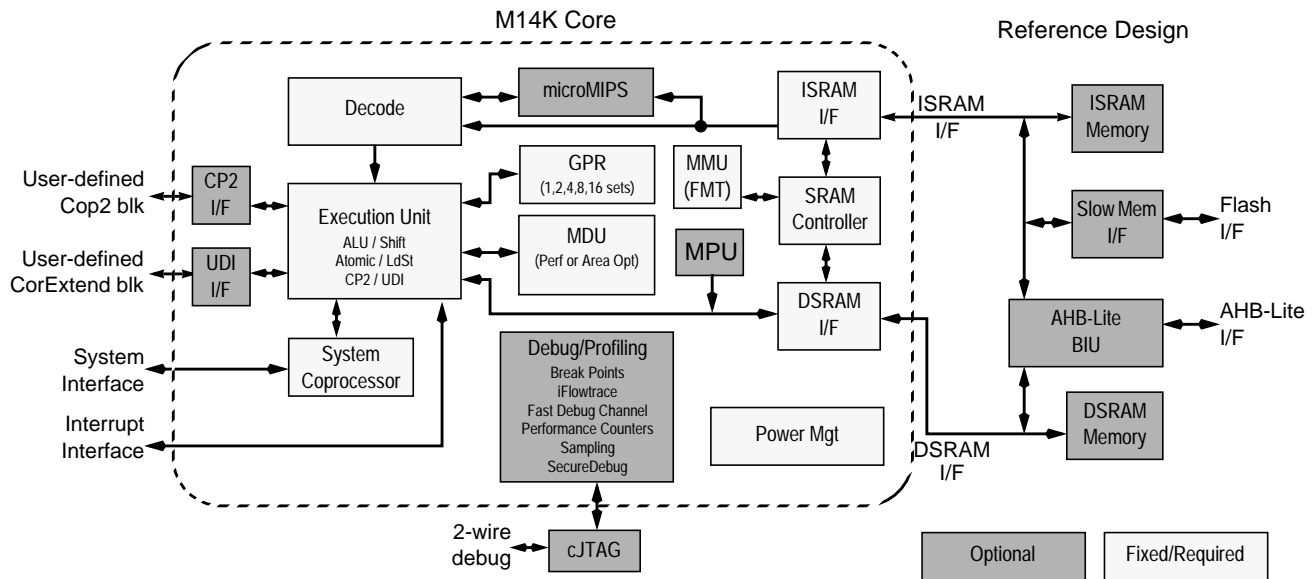
## 1.2 M14K™ Core Block Diagram

The M14K core contains both required and optional blocks, as shown in the block diagram in [Figure 1](#). Required blocks are the lightly shaded areas of the block diagram and are always present in any core implementation. Optional blocks may be added to the base core, depending on the needs of a specific implementation. The required blocks are as follows:

- Execution Unit
- General Proposed Registers (GPR)
- Multiply-Divide Unit (MDU)
- System Control Coprocessor (CP0)
- Memory Management Unit (MMU)
- SRAM Interface
- Power Management

Optional blocks include:

- Reference Design
- microMIPS Instruction Recode
- EJTAG Debug Support
- Coprocessor 2 Interface (CP2)
- CorExtend® User Defined Instructions (UDI)

**Figure 1 M14K™ Processor Core Block Diagram**

### 1.2.1 Required Logic Blocks

The following subsections describe the required logic blocks of the M14K processor core.

#### 1.2.1.1 Execution Unit

The core execution unit implements a load-store architecture with single-cycle Arithmetic Logic Unit (ALU) operations (logical, shift, add, subtract) and an autonomous multiply-divide unit.

The execution unit includes:

- 32-bit adder used for calculating the data address
- Address unit for calculating the next instruction address
- Logic for branch determination and branch target address calculation
- Load aligner
- Bypass multiplexers used to avoid stalls when executing instruction streams where data-producing instructions are followed closely by consumers of their results
- Leading Zero/One detect unit for implementing the CLZ and CLO instructions
- Arithmetic Logic Unit (ALU) for performing bitwise logical operations
- Shifter and Store aligner
-

### 1.2.1.2 General Purposed Register (GPR) Shadow Registers

The M14K core contains thirty-two 32-bit general-purpose registers used for integer operations and address calculation. Optionally, one, three, seven or fifteen additional register file shadow sets (each containing thirty-two registers) can be added to minimize context switching overhead during interrupt/exception processing. The register file consists of two read ports and one write port and is fully bypassed to minimize operation latency in the pipeline.

### 1.2.1.3 Multiply/Divide Unit (MDU)

The Multiply/Divide unit performs multiply and divide operations. Two configuration options exist for the MDU, selectable at build time: an area-efficient iterative MDU and a higher performance 32x16 array. The MDU consists of an iterative or 32x16 Booth-encoded multiplier, result-accumulation registers (HI and LO), multiply and divide state machines, and all multiplexers and control logic required to perform these functions. The high-performance pipelined MDU supports execution of a 16x16 or 32x16 multiply operation every clock cycle; 32x32 multiply operations can be issued every other clock cycle. Appropriate interlocks are implemented to stall the issue of back-to-back 32x32 multiply operations. Divide operations are implemented with a simple 1 bit per clock iterative algorithm and require 35 clock cycles in worst case to complete. Early-in to the algorithm detects sign extension of the dividend, if it is actual size is 24, 16 or 8 bit. the divider will skip 7, 15 or 23 of the 32 iterations. An attempt to issue a subsequent MDU instruction while a divide is still active causes a pipeline stall until the divide operation is completed.

The area-efficient, non-pipelined MDU consists of a 32-bit full-adder, result-accumulation registers (HI and LO), a combined multiply/divide state machine, and all multiplexers and control logic required to perform these functions. It performs any multiply using 32 cycles in an iterative 1 bit per clock algorithm. Divide operations are also implemented with a simple 1 bit per clock iterative algorithm (no early-in) and require 35 clock cycles to complete. An attempt to issue a subsequent MDU instruction while a multiply/divide is still active causes a pipeline stall until the operation is completed.

The M14K implements an additional multiply instruction, MUL, which specifies that lower 32-bits of the multiply result be placed in the register file instead of the HI/LO register pair. By avoiding the explicit move from LO (MFLO) instruction, required when using the LO register, and by supporting multiple destination registers, the throughput of multiply-intensive operations is increased.

Two instructions, multiply-add (MADD/MADDU) and multiply-subtract (MSUB/MSUBU), are used to perform the multiply-add and multiply-subtract operations. The MADD instruction multiplies two numbers and then adds the product to the current contents of the HI and LO registers. Similarly, the MSUB instruction multiplies two operands and then subtracts the product from the HI and LO registers. The MADD/MADDU and MSUB/MSUBU operations are commonly used in Digital Signal Processor (DSP) algorithms.

### 1.2.1.4 System Control Coprocessor (CP0)

In the MIPS architecture, CP0 is responsible for the virtual-to-physical address translation, the exception control system, the processor's diagnostics capability, operating mode selection (kernel vs. user mode), and the enabling/disabling of interrupts. Configuration information such as presence of build-time options like microMIPS, CorExtend ASE or Coprocessor 2 interface, is also available by accessing the CP0 registers.

Coprocessor 0 also contains the logic for identifying and managing exceptions. Exceptions can be caused by a variety of sources, including boundary cases in data, external events, or program errors.

#### ***Interrupt Handling***

The M14K core includes support for eight hardware interrupt pins, two software interrupts, and a timer interrupt. These interrupts can be used in any of three interrupt modes, as defined by Release 2 of the MIPS32 Architecture:

- Interrupt compatibility mode, which acts identically to that in an implementation of Release 1 of the Architecture.
- Vectored Interrupt (VI) mode, which adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt, and to assign a GPR shadow set for use during interrupt processing. The presence of this mode is denoted by the VInt bit in the *Config3* register. This mode is architecturally optional; but it is always present on the M14K core, so the VInt bit will always read as a 1 for the M14K core.
- External Interrupt Controller (EIC) mode, which redefines the way in which interrupts are handled to provide full support for an external interrupt controller handling prioritization and vectoring of interrupts. The presence of this mode denoted by the VEIC bit in the *Config3* register. Again, this mode is architecturally optional. On the M14K core, the VEIC bit is set externally by the static input, *SI\_EICPresent*, to allow system logic to indicate the presence of an external interrupt controller.

The reset state of the processor is interrupt compatibility mode, such that a processor supporting Release 2 of the Architecture, the M14K core for example, is fully compatible with implementations of Release 1 of the Architecture.

VI or EIC interrupt modes can be combined with the optional shadow registers to specify which shadow set should be used on entry to a particular vector. The shadow registers further improve interrupt latency by avoiding the need to save context when invoking an interrupt handler.

In the M14K core, interrupt latency is greatly improved over the M4K by:

- Speculative interrupt vector prefetching during the pipeline flush
- Interrupt Automated Prologue (IAP) by hardware: Shadow Register Sets remove the need to save GPRs, and IAP removes the need to save specific Control Registers when handling an interrupt.
- Interrupt Automated Epilogue (IAE) by hardware: Shadow Register Sets remove the need to restore GPRs, and IAE removes the need to restore specific Control Registers when returning from an interrupt.
- Allow interrupt chaining. When servicing an interrupt and interrupt chaining is enabled, there is no need to return from the current Interrupt Service Routine (ISR) if there is another valid interrupt pending to be serviced. The control of the processor can jump directly from the current ISR to the next ISR without IAE and IAP.

Refer to [Chapter 5, “CP0 Registers of the M14K™ Core”](#) on page 95 for more information on the CP0 registers. Refer to [Chapter 8, “EJTAG Debug Support in the M14K™ Core”](#) on page 159 for more information on EJTAG debug registers.

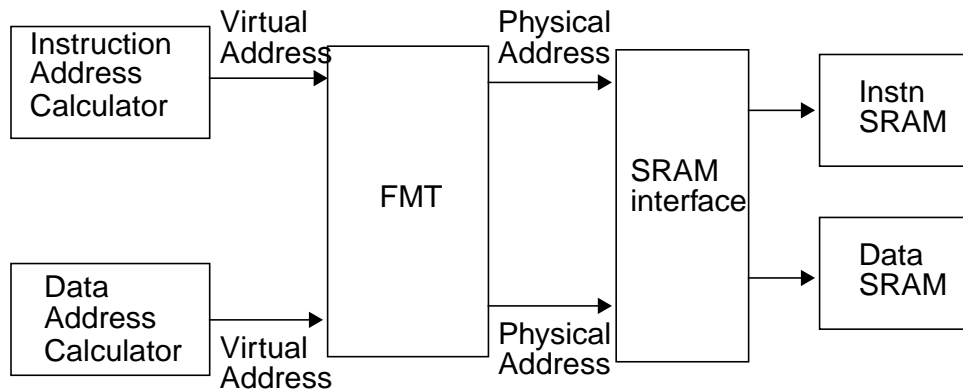
### 1.2.1.5 Memory Management Unit (MMU)

The M14K core contains an MMU that interfaces between the execution unit and the SRAM controller, shown in [Figure 1.1](#).

The M14K implements an FMT-based MMU. The FMT performs a simple translation to obtain the physical address from the virtual address. Refer to [Chapter 3, “Memory Management of the M14K™ Core”](#) on page 51 for more information on the FMT.

[Figure 1.1](#) shows how the address translation mechanism interacts with SRAM access.

Figure 1.1 Address Translation During a SRAM Access



#### 1.2.1.6 SRAM Interface Controller

Instead of caches, the M14K core contains an interface to SRAM-style memories that can be tightly coupled to the core. This permits deterministic response time with less area than is typically required for caches. The SRAM interface includes separate uni-directional 32-bit buses for address, read data, and write data.

##### ***Dual or Unified Interfaces***

The SRAM interface includes a build-time option to select either dual or unified instruction and data interfaces.

The dual interface enables independent connection to instruction and data devices. It generally yields the highest performance, because the pipeline can generate simultaneous I and D requests, which are then serviced in parallel.

For simpler or cost-sensitive systems, it is also possible to combine the I and D interfaces into a common interface that services both types of requests. If I and D requests occur simultaneously, priority is given to the D side.

##### ***Back-stalling***

Typically, read and write transactions will complete in a single cycle. However, if multi-cycle latency is desired, the interface can be stalled to allow connection to slower devices.

##### ***Redirection***

When the dual I/D interface is present, a mechanism exists to divert D-side references to the I-side, if desired. The mechanism can be explicitly invoked for any other D-side references, as well. When the *DS\_Redir* signal is asserted, a D-side request is diverted to the I-side interface in the following cycle, and the D-side will be stalled until the transaction is completed.

##### ***Transaction Abort***

The core may request a transaction (fetch/load/store/sync) to be aborted. This is particularly useful in case of interrupts. Because the core does not know whether transactions are re-startable, it cannot arbitrarily interrupt a request which has been initiated on the SRAM interface. However, cycles spent waiting for a multi-cycle transaction to complete can directly impact interrupt latency. In order to minimize this effect, the interface supports an abort mechanism. The core requests an abort whenever an interrupt is detected and a transaction is pending (abort of an instruction fetch may also be requested in other cases). The external system logic can choose to acknowledge or to ignore the abort request.

### ***Connecting to Narrower Devices***

The instruction and data read buses are always 32 bits in width. To facilitate connection to narrower memories, the SRAM interface protocol includes input byte enables that can be used by system logic to signal validity as partial read data becomes available. The input byte enables conditionally register the incoming read data bytes within the core, and thus eliminate the need for external registers to gather the entire 32 bits of data. External muxes are required to redirect the narrower data to the appropriate byte lanes.

### ***Lock Mechanism***

The SRAM interface includes a protocol to identify a locked sequence, and is used in conjunction with the LL/SC atomic read-modify-write semaphore instructions.

### ***Sync Mechanism***

The interface includes a protocol that externalizes the execution of the SYNC instruction. External logic might choose to use this information to enforce memory ordering between various elements in the system.

### ***External Call Indication***

The instruction fetch interface contains signals that indicate that the core is fetching the target of a subroutine call-type instruction such as JAL or BAL. At some point after a call, there will typically be a return to the original code sequence. If a system prefetches instructions, it can make use of this information to save instructions that were prefetched and are likely to be executed after the return.

#### **1.2.1.7 Power Management**

The core offers a number of power management features, including low-power design, active power management, and power-down modes of operation. The core is a static design that supports a WAIT instruction designed to signal the rest of the device that execution and clocking should be halted, hence reducing system power consumption during idle periods.

The core provides two mechanisms for system-level, low-power support:

- Register-controlled power management
- Instruction-controlled power management

In register-controlled power management mode the core provides three bits in the CP0 Status register for software control of the power management function and allows interrupts to be serviced even when the core is in power-down mode. In instruction-controlled power-down mode execution of the WAIT instruction is used to invoke low-power mode.

Refer to [Chapter 7, “Power Management of the M14K™ Core”](#) on page 157 for more information on power management.

## **1.2.2 Optional Logic Blocks**

The core consists of the following optional logic blocks as shown in the block diagram in [Figure 1](#).

#### **1.2.2.1 Reference Design**

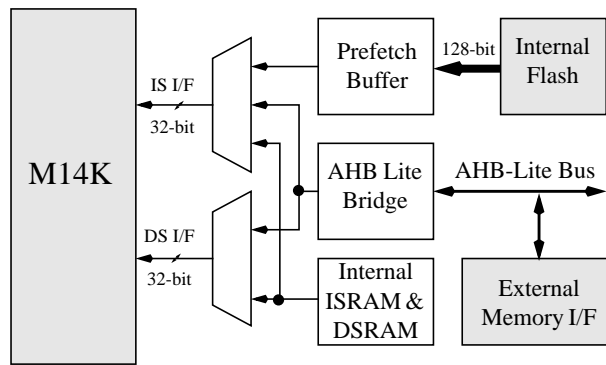
The M14K core contains a reference design that shows a typical usage of the core with:



- Dual I-SRAM and D-SRAM interface with fast memories (i.e., SRAM) for instruction and data storage.
- Optimized interface for slow memory (i.e., Flash memory) access by having a prefetch buffer and a wider Data Read bus (i.e., IS\_RData[127:0]) to speed up I-Fetch performance.
- AHB-lite bus interface to the system bus if the memory accesses are outside the memory map for the SRAM and Flash regions. AHB-Lite is a subset of the AHB bus protocol that supports a single bus master. The interface shares the same 32-bit Read and Write address bus and has two unidirectional 32-bit buses for Read and Write data.

The reference design is optional and can be modified by the user to better fit the SOC design requirement.

**Figure 1.2 Reference Design Block Diagram.**



### 1.2.2.2 microMIPS Instruction Recode/Decode

The M14K core supports the microMIPS instruction set, which contains all MIPS32 ISA instructions (except for branch-likely instructions) in a new, 32-bit encoding scheme, with some of the commonly used instructions also available in 16-bit encoded format. This ISA improves code density through the additional 16-bit instructions, while maintaining a performance similar to MIPS32 mode. In microMIPS mode, 16-bit or 32-bit instructions are fetched and recoded to legacy MIPS32 instruction opcodes in the pipeline's I stage, so that the M14K core can have the same M4K microarchitecture. Because the microMIPS instruction stream can be intermixed with 16-bit halfword or 32-bit word size instructions on halfword or word boundaries, additional logic is in place to address the word misalignment issues, thus minimizing performance loss.

### 1.2.2.3 Memory Protection Unit (MPU)

The MPU can be configured to have from 1 to 16 memory protection regions. Each region is enabled by a set of Watch registers that define the address, size, and protection of each memory region. The MPU control and Watch registers are implemented by CDMM (Common Device Memory Map) registers. After they have been programmed, these control registers can be locked to prohibit later modifications. Once programmed, a Protection Exception will be triggered when an Instruction Fetch or Data Access matches the address of the protected memory region or any modification of the EBase (base address of exception vectors) register was attempted. Each protected region can also disable the iFlowtrace capability. Typically, the MPU improves system security by disabling access to bootcode and preventing execution of non-trusted code executing in kernel mode.

### 1.2.2.4 EJTAG Debug Support

The M14K core provides for an optional Enhanced JTAG (EJTAG) interface for use in the software debug of application and kernel code. In addition to standard user mode and kernel modes of operation, the M14K core provides a Debug mode that is entered after a debug exception (derived from a hardware breakpoint, single-step exception, etc.) is taken and continues until a debug exception return (DERET) instruction is executed. During this time, the processor executes the debug exception handler routine.

The EJTAG interface operates through the Test Access Port (TAP), a serial communication port used for transferring test data in and out of the M14K core. In addition to the standard JTAG instructions, special instructions defined in the EJTAG specification specify which registers are selected and how they are used.

#### ***Debug Registers***

Four debug registers (*DEBUG*, *DEBUG2*, *DEPC*, and *DESAVE*) have been added to the MIPS Coprocessor 0 (CP0) register set. The *DEBUG* and *DEBUG2* registers show the cause of the debug exception and are used for setting up single-step operations. The *DEPC* (Debug Exception Program Counter) register holds the address on which the debug exception was taken, which is used to resume program execution after the debug operation finishes. Finally, the *DESAVE* (Debug Exception Save) register enables the saving of general-purpose registers used during execution of the debug exception handler.

To exit debug mode, a Debug Exception Return (DERET) instruction is executed. When this instruction is executed, the system exits debug mode, allowing normal execution of application and system code to resume.

#### ***EJTAG Hardware Breakpoints***

There are several types of *simple* hardware breakpoints defined in the EJTAG specification. These stop the normal operation of the CPU and force the system into debug mode. There are two types of simple hardware breakpoints implemented in the M14K core: Instruction breakpoints and Data breakpoints. Additionally, *complex* hardware breakpoints can be included, which allow detection of more intricate sequences of events.

The M14K core can be configured with the following breakpoint options:

- No data, instruction, or complex breakpoints
- One data and two instruction breakpoints, without complex breakpoints
- Two data and four instruction breakpoints, without complex breakpoints
- Two data and six instruction breakpoints, with or without complex breakpoints
- Four data and eight instruction breakpoints, with or without complex breakpoints

Instruction breakpoints occur on instruction execution operations, and the breakpoint is set on the virtual address. A mask can be applied to the virtual address to set breakpoints on a binary range of instructions.

Data breakpoints occur on load/store transactions. The breakpoint is set on a virtual address value, with the same single address or binary address range as the Instruction breakpoint. Data breakpoints can be set on a load, a store, or both. Data breakpoints can also be set to match on the operand value of the load/store operation, with byte-granularity masking. Finally, masks can be applied to both the virtual address and the load/store value.

Complex breakpoints utilize the simple instruction and data breakpoints and break when a combination of events is seen. Complex break features include:

- **Pass Counters** - Each time a matching condition is seen, a counter is decremented. The break or trigger will only be enabled when the counter has counted down to 0.
- **Tuples** - A tuple is the pairing of an instruction and a data breakpoint. The tuple will match if both the virtual address of the load or store instruction matches the instruction breakpoint, and the data breakpoint of the resulting load or store address and optional data value matches.
- **Priming** - This allows a breakpoint to be enabled only after other break conditions have been met. Also called sequential or armed triggering.
- **Qualified** - This feature uses a data breakpoint to qualify when an instruction breakpoint can be taken. Once a load matches the data address and the data value, the instruction break will be enabled. If a load matches the address, but has mis-matching data, the instruction break will be disabled.

### Performance Counters

Performance counters are used to accumulate occurrences of internal predefined events/cycles/conditions for program analysis, debug, or profiling. A few examples of event types are clock cycles, instructions executed, specific instruction types executed, loads, stores, exceptions, and cycles while the CPU is stalled. There are two, 32-bit counters. Each can count one of the 64 internal predefined events selected by a corresponding control register. A counter overflow can be programmed to generate an interrupt, where the interrupt handler software can maintain larger total counts.

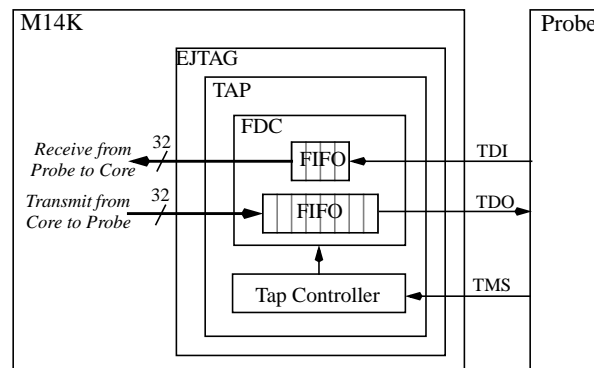
### PC/Address Sampling

This sampling function is used for program profiling and hot-spots analysis. Instruction PC and/or Load/Store addresses can be sampled periodically. The result is scanned out through the EJTAG port. The Debug Control Register (*DCR*) is used to specify the sample period and the sample trigger.

### Fast Debug Channel (FDC)

The M14K core includes optional FDC as a mechanism for high bandwidth data transfer between a debug host/probe and a target. FDC provides a FIFO buffering scheme to transfer data serially, with low CPU overhead and minimized waiting time. The data transfer occurs in the background, and the target CPU can either choose to check the status of the transfer periodically, or it can choose to be interrupted at the end of the transfer.

**Figure 1.3 FDC Overview**



**iFlowtrace™**

The M14K core has an option for a simple trace mechanism called iFlowtrace. This mechanism traces only the instruction PC, not data addresses or values. This simplification allows the trace block to be small and the trace compression to be very efficient. iFlowtrace memory can be configured as off-chip, on-chip, or both.

iFlowtrace also offers special-event trace modes when normal tracing is disabled, namely:

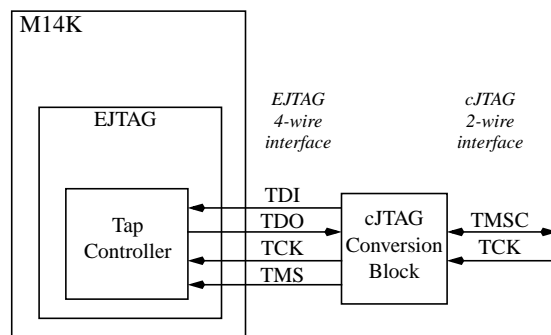
- Function Call/Return and Exception Tracing mode to trace the PC value of function calls and returns and/or exceptions and returns.
- Breakpoint Match mode traces the breakpoint ID of a matching breakpoint and, for data breakpoints, the PC value of the instruction that caused it.
- Filtered Data Tracing mode traces the ID of a matching data breakpoint, the load or store data value, access type and memory access size, and the low-order address bits of the memory access, which is useful when the data breakpoint is set to match a binary range of addresses.
- User Trace Messages. The user can instrument their code to add their own 32-bit value messages into the trace by writing to the CP0 *User Trace Data* register.
- Delta Cycle mode works in combination with the above trace modes to provide a timestamp between stored events. It reports the number of cycles that have elapsed since the last message was generated and put into the trace.

Refer to [Chapter 8, “EJTAG Debug Support in the M14K™ Core”](#) on page 159 for more information on the EJTAG features.

**cJTAG Support**

The M14K core provides an external conversion block that converts the existing EJTAG (IEEE 1149.1) 4-wire interface at the M14K core to a cJTAG (IEEE 1149.7) 2-wire interface. cJTAG reduces the number of wires from 4 to 2 and enables the support of Star-2 scan topology in the system debug environment.

**Figure 1.4 cJTAG Support**

**SecureDebug**

SecureDebug improves security by disabling untrusted EJTAG debug access. An input signal is used to disable debug features, such as Probe Trap, Debug Interrupt Exception (EjtagBrk and DINT signal), the EJTAGBOOT instruction, and PC Sampling.

### 1.2.2.5 Coprocessor 2 Interface (CP2)

The optional coprocessor 2 (CP2) interface provides a full-featured interface for a coprocessor. It provides full support for all the MIPS32 COP2 instructions, with the exception of the 64-bit Load/Store instructions (LDC2/SDC2).

The CP2 interface can provide access to a graphics accelerator coprocessor or a simple register file. There is no support for the floating-point coprocessor COP1, which requires 64-bit data transfers.

Refer to [Chapter 10, “M14K™ Processor Core Instructions”](#) on page 239 for more information on the Coprocessor 2 supported instructions.

### 1.2.2.6 CorExtend® User Defined Instructions (UDI)

This optional module contains support for CorExtend user defined instructions. These instructions must be defined at build-time for the M14K core. Access to UDI requires a separate license from MIPS, and the core is then referred to as the M14K Pro™ core. When licensed, 16 instructions in the opcode map are available for UDI, and each instruction can have single or multi-cycle latency. A UDI instruction can operate on any one or two general-purpose registers or immediate data contained within the instruction, and can write the result of each instruction back to a general purpose register or local register. Implementation details for UDI can be found in other documents available from MIPS.

Refer to [Table 10.3 “Special2 Opcode Encoding of Function Field”](#) for a specification of the opcode map available for user defined instructions.



## Pipeline of the M14K™ Core

The M14K processor core implements a 5-stage pipeline similar to the original R3000 pipeline. The pipeline allows the processor to achieve high frequency while minimizing device complexity, reducing both cost and power consumption. This chapter contains the following sections:

- [Section 2.1 “Pipeline Stages”](#)
- [Section 2.2 “Multiply/Divide Operations”](#)
- [Section 2.3 “MDU Pipeline \(High-Performance MDU\)”](#)
- [Section 2.4 “MDU Pipeline \(Area-Efficient MDU\)”](#)
- [Section 2.5 “Branch Delay”](#)
- [Section 2.6 “Data Bypassing”](#)
- [Section 2.8 “Interlock Handling”](#)
- [Section 2.9 “Slip Conditions”](#)
- [Section 2.10 “Instruction Interlocks”](#)
- [Section 2.11 “Hazards”](#)

### 2.1 Pipeline Stages

The pipeline consists of five stages:

- Instruction (I stage)
- Execution (E stage)
- Memory (M stage)
- Align (A stage)
- Writeback (W stage)

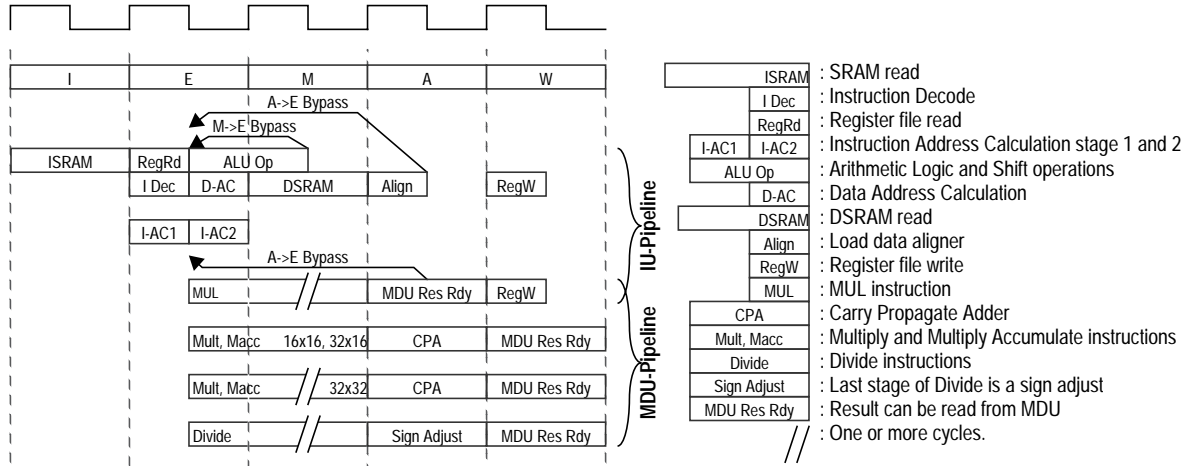
The M14K core implements a “Bypass” mechanism that allows the result of an operation to be sent directly to the instruction that needs it, without having to write the result to the register and then read it back.

The M14K soft core includes a build-time option that determines the type of multiply/divide unit (MDU) implemented. The MDU can be either a high-performance array or an iterative, area-efficient array. The MDU choice has a

significant effect on the MDU pipeline, and the latency of multiply/divide instructions executed on the core. Software can query the type of MDU present on a specific implementation of the core by querying the MDU bit in the Config register (CP0 register 16, select 0); see [Chapter 5, “CP0 Registers of the M14K™ Core”](#) on page 95 for more details.

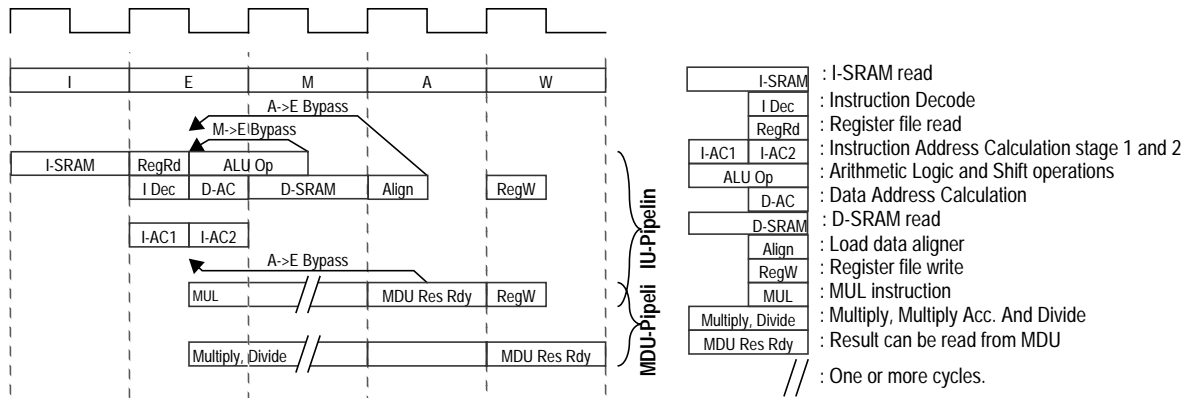
[Figure 2.1](#) shows the operations performed in each pipeline stage of the M14K processor core, when the high-performance multiplier is present.

**Figure 2.1 M14K™ Core Pipeline Stages (with high-performance MDU)**



[Figure 2.2](#) shows the operations performed in each pipeline stage of the M14K processor core, when the area-efficient multiplier is present.

**Figure 2.2 M14K™ Core Pipeline Stages (with area-efficient MDU)**



### 2.1.1 I Stage: Instruction Fetch

During the Instruction fetch stage:

- An instruction is fetched from the instructionSRAM.



- If both MIPS32 and microMIPS ISAs are supported, microMIPS instructions are converted to MIPS32-like instructions. If the MIPS32 ISA is not supported, 16-bit microMIPS instructions will be first recoded into 32-bit microMIPS equivalent instructions, and then decoded in native microMIPS ISA format.

### 2.1.2 E Stage: Execution

During the Execution stage:

- Operands are fetched from the register file.
- Operands from the M and A stage are bypassed to this stage.
- The Arithmetic Logic Unit (ALU) begins the arithmetic or logical operation for register-to-register instructions.
- The ALU calculates the data virtual address for load and store instructions and the MMU performs the fixed virtual-to-physical address translation.
- The ALU determines whether the branch condition is true and calculates the virtual branch target address for branch instructions.
- Instruction logic selects an instruction address and the MMU performs the fixed virtual-to-physical address translation.
- All multiply and divide operations begin in this stage.

### 2.1.3 M Stage: Memory Fetch

During the Memory fetch stage:

- The arithmetic ALU operation completes.
- The data SRAM access is performed for load and store instructions.
- A 16x16 or 32x16 multiply calculation completes (high-performance MDU option).
- A 32x32 multiply operation stalls the MDU pipeline for one clock in the M stage (high-performance MDU option).
- A multiply operation stalls the MDU pipeline for 31 clocks in the M stage (area-efficient MDU option).
- A multiply-accumulate operation stalls the MDU pipeline for 33 clocks in the M stage (area-efficient MDU option).
- A divide operation stalls the MDU pipeline for a maximum of 34 clocks in the M stage. Early-in sign extension detection on the dividend will skip 7, 15, or 23 stall clocks (only the divider in the fast MDU option supports early-in detection).

### 2.1.4 A Stage: Align

During the Align stage:

- Load data is aligned to its word boundary.

- A multiply/divide operation updates the HI/LO registers (area-efficient MDU option).
- Multiply operation performs the carry-propagate-add. The actual register writeback is performed in the W stage (high-performance MDU option).
- A MUL operation makes the result available for writeback. The actual register writeback is performed in the W stage.
- EJTAG complex break conditions are evaluated.

### 2.1.5 W Stage: Writeback

During the Writeback stage:

- For register-to-register or load instructions, the result is written back to the register file.

## 2.2 Multiply/Divide Operations

The M14K core implement the standard MIPS II™ multiply and divide instructions. Additionally, several new instructions were standardized in the MIPS32 architecture for enhanced performance.

The targeted multiply instruction, MUL, specifies that multiply results be placed in the general purpose register file instead of the HI/LO register pair. By avoiding the explicit MFLO instruction, required when using the LO register, and by supporting multiple destination registers, the throughput of multiply-intensive operations is increased.

Four instructions, multiply-add (MADD), multiply-add-unsigned (MADDU) multiply-subtract (MSUB), and multiply-subtract-unsigned (MSUBU), are used to perform the multiply-accumulate and multiply-subtract operations. The MADD/MADDU instruction multiplies two numbers and then adds the product to the current contents of the HI and LO registers. Similarly, the MSUB/MSUBU instruction multiplies two operands and then subtracts the product from the HI and LO registers. The MADD/MADDU and MSUB/MSUBU operations are commonly used in DSP algorithms.

All multiply operations (except the MUL instruction) write to the HI/LO register pair. All integer operations write to the general purpose registers (GPR). Because MDU operations write to different registers than integer operations, following integer instructions can execute before the MDU operation has completed. The MFLO and MFHI instructions are used to move data from the HI/LO register pair to the GPR file. If a MFLO or MFHI instruction is issued before the MDU operation completes, it will stall to wait for the data.

## 2.3 MDU Pipeline (High-Performance MDU)

The M14K processor core contains an autonomous multiply/divide unit (MDU) with a separate pipeline for multiply and divide operations. This pipeline operates in parallel with the integer unit (IU) pipeline and does not stall when the IU pipeline stalls. This allows multi-cycle MDU operations, such as a divide, to be partially masked by system stalls and/or other integer unit instructions.

The MDU consists of a 32x16 Booth-encoded multiplier array, a carry propagate adder, result/accumulation registers (HI and LO), multiply and divide state machines, and all necessary multiplexers and control logic. The first number shown ('32' of 32x16) represents the *rs* operand. The second number ('16' of 32x16) represents the *rt* operand. The core only checks the latter (*rt*) operand value to determine how many times the operation must pass through the multiplier array. The 16x16 and 32x16 operations pass through the multiplier array once. A 32x32 operation passes through the multiplier array twice.

The MDU supports execution of a 16x16 or 32x16 multiply operation every clock cycle; 32x32 multiply operations can be issued every other clock cycle. Appropriate interlocks are implemented to stall the issue of back-to-back 32x32 multiply operations. Multiply operand size is automatically determined by logic built into the MDU. Divide operations are implemented with a simple 1 bit per clock iterative algorithm with an early in detection of sign extension on the dividend (*rs*). Any attempt to issue a subsequent MDU instruction while a divide is still active causes an IU pipeline stall until the divide operation is completed.

Table 2.1 lists the latencies (number of cycles until a result is available) for multiply, and divide instructions. The latencies are listed in terms of pipeline clocks. In this table ‘latency’ refers to the number of cycles necessary for the first instruction to produce the result needed by the second instruction.

**Table 2.1 MDU Instruction Latencies (High-Performance MDU)**

Size of Operand 1st Instruction <sup>[1]</sup>	Instruction Sequence		Latency Clocks
	1st Instruction	2nd Instruction	
16 bit	MULT/MULTU, MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU or MFHI/MFLO	1
32 bit	MULT/MULTU, MADD/MADDU, or MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU or MFHI/MFLO	2
16 bit	MUL	Integer operation <sup>[2]</sup>	2 <sup>[3]</sup>
32 bit	MUL	Integer operation <sup>[2]</sup>	2 <sup>[3]</sup>
8 bit	DIVU	MFHI/MFLO	9
16 bit	DIVU	MFHI/MFLO	17
24 bit	DIVU	MFHI/MFLO	25
32 bit	DIVU	MFHI/MFLO	33
8 bit	DIV	MFHI/MFLO	10 <sup>[4]</sup>
16 bit	DIV	MFHI/MFLO	18 <sup>[4]</sup>
24 bit	DIV	MFHI/MFLO	26 <sup>[4]</sup>
32 bit	DIV	MFHI/MFLO	34 <sup>[4]</sup>
any	MFHI/MFLO	Integer operation <sup>[2]</sup>	2
any	MTHI/MTLO	MADD/MADDU or MSUB/MSUBU	1
<p>[1] For multiply operations, this is the <i>rt</i> operand. For divide operations, this is the <i>rs</i> operand.  [2] Integer Operation refers to any integer instruction that uses the result of a previous MDU operation.  [3] This does not include the 1 or 2 IU pipeline stalls (16 bit or 32 bit) that the MUL operation causes irrespective of the following instruction. These stalls do not add to the latency of 2.  [4] If both operands are positive, then the Sign Adjust stage is bypassed. Latency is then the same as for DIVU.</p>			

In Table 2.1, a latency of one means that the first and second instructions can be issued back-to-back in the code, without the MDU causing any stalls in the IU pipeline. A latency of two means that if issued back-to-back, the IU pipeline will be stalled for one cycle. MUL operations are special, because the MDU needs to stall the IU pipeline in order to maintain its register file write slot. As a result, the MUL 16x16 or 32x16 operation will always force a one-cycle stall of the IU pipeline, and the MUL 32x32 will force a two-cycle stall. If the integer instruction immediately following the MUL operation uses its result, an additional stall is forced on the IU pipeline.

Table 2.2 lists the repeat rates (peak issue rate of cycles until the operation can be reissued) for multiply accumulate/subtract instructions. The repeat rates are listed in terms of pipeline clocks. In this table ‘repeat rate’ refers to the case where the first MDU instruction (in the table below) is back-to-back with the second instruction.

**Table 2.2 MDU Instruction Repeat Rates (High-Performance MDU)**

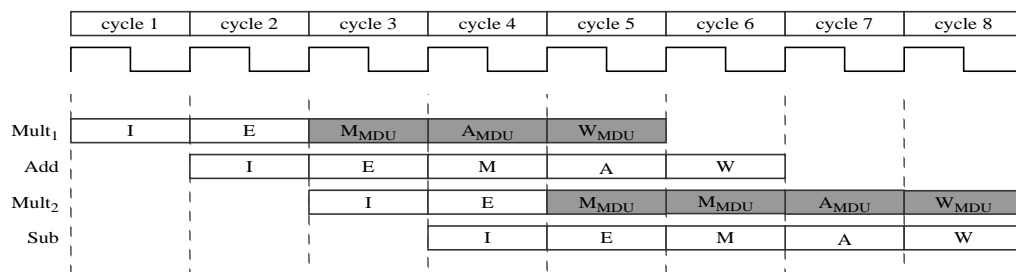
Operand Size of 1st Instruction	Instruction Sequence		Repeat Rate
	1st Instruction	2nd Instruction	
16 bit	MULT/MULTU, MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU	1
32 bit	MULT/MULTU, MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU	2

Figure 2.3 below shows the pipeline flow for the following sequence:

1. 32x16 multiply (Mult<sub>1</sub>)
2. Add
3. 32x32 multiply (Mult<sub>2</sub>)
4. Subtract (Sub)

The 32x16 multiply operation requires one clock of each pipeline stage to complete. The 32x32 multiply operation requires two clocks in the M<sub>MDU</sub> pipe-stage. The MDU pipeline is shown as the shaded areas of Figure 2.3 and always starts a computation in the final phase of the E stage. As shown in the figure, the M<sub>MDU</sub> pipe-stage of the MDU pipeline occurs in parallel with the M stage of the IU pipeline, the A<sub>MDU</sub> stage occurs in parallel with the A stage, and the W<sub>MDU</sub> stage occurs in parallel with the W stage. In general this need not be the case. Following the 1st cycle of the M stages, the two pipelines need not be synchronized. This does not present a problem because results in the MDU pipeline are written to the HI and LO registers, while the integer pipeline results are written to the register file.

**Figure 2.3 MDU Pipeline Behavior During Multiply Operations**



The following is a cycle-by-cycle analysis of Figure 2.3.

1. The first 32x16 multiply operation (Mult<sub>1</sub>) is fetched from the instruction cache and enters the I stage.

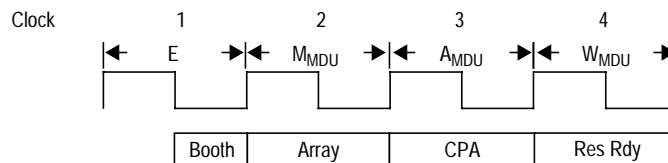
2. An Add operation enters the I stage. The Mult<sub>1</sub> operation enters the E stage. The integer and MDU pipelines share the I and E pipeline stages. At the end of the E stage in cycle 2, the MDU pipeline starts processing the multiply operation (Mult<sub>1</sub>).
3. In cycle 3, a 32x32 multiply operation (Mult<sub>2</sub>) enters the I stage and is fetched from the instruction cache. Since the Add operation has not yet reached the M stage by cycle 3, there is no activity in the M stage of the integer pipeline at this time.
4. In cycle 4, the Subtract instruction enters I stage. The second multiply operation (Mult<sub>2</sub>) enters the E stage. And the Add operation enters M stage of the integer pipe. Since the Mult<sub>1</sub> multiply is a 32x16 operation, only one clock is required for the M<sub>MDU</sub> stage, hence the Mult<sub>1</sub> operation passes to the A<sub>MDU</sub> stage of the MDU pipeline.
5. In cycle 5, the Subtract instruction enters E stage. The Mult<sub>2</sub> multiply enters the M<sub>MDU</sub> stage. The Add operation enters the A stage of the integer pipeline. The Mult<sub>1</sub> operation completes and is written back in to the HI/LO register pair in the W<sub>MDU</sub> stage.
6. Since a 32x32 multiply requires two passes through the multiplier, with each pass requiring one clock, the 32x32 Mult<sub>2</sub> remains in the M<sub>MDU</sub> stage in cycle 6. The Sub instruction enters M stage in the integer pipeline. The Add operation completes and is written to the register file in the W stage of the integer pipeline.
7. The Mult<sub>2</sub> multiply operation progresses to the A<sub>MDU</sub> stage, and the Sub instruction progress to the A stage.
8. The Mult<sub>2</sub> operation completes and is written to the HI/LO registers pair in the the W<sub>MDU</sub> stage, while the Sub instruction writes to the register file in the W stage.

### 2.3.1 32x16 Multiply (High-Performance MDU)

The 32x16 multiply operation begins in the last phase of the E stage, which is shared between the integer and MDU pipelines. In the latter phase of the E stage, the *rs* and *rt* operands arrive and the Booth-recoding function occurs at this time. The multiply calculation requires one clock and occurs in the M<sub>MDU</sub> stage. In the A<sub>MDU</sub> stage, the carry-propagate-add (CPA) function occurs and the operation is completed. The result is ready to be read from the HI/LO registers in the W<sub>MDU</sub> stage.

Figure 2.4 shows a diagram of a 32x16 multiply operation.

**Figure 2.4 MDU Pipeline Flow During a 32x16 Multiply Operation**

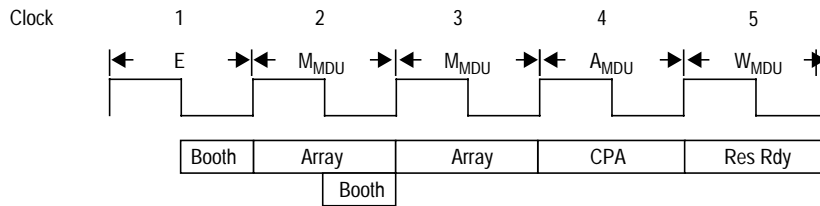


### 2.3.2 32x32 Multiply (High-Performance MDU)

The 32x32 multiply operation begins in the last phase of the E stage, which is shared between the integer and MDU pipelines. In the latter phase of the E stage, the *rs* and *rt* operands arrive and the Booth-recoding function occurs at this time. The multiply calculation requires two clocks and occurs in the M<sub>MDU</sub> stage. In the A<sub>MDU</sub> stage, the CPA function occurs and the operation is completed.

Figure 2.5 shows a diagram of a 32x32 multiply operation.

**Figure 2.5 MDU Pipeline Flow During a 32x32 Multiply Operation**



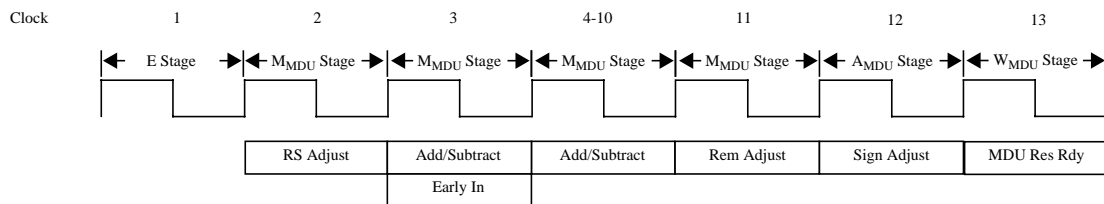
### 2.3.3 Divide (High-Performance MDU)

Divide operations are implemented using a simple non-restoring division algorithm. This algorithm works only for positive operands, hence the first cycle of the  $M_{MDU}$  stage is used to negate the  $rs$  operand (RS Adjust) if needed. Note that this cycle is spent even if the adjustment is not necessary. During the next maximum 32 cycles (3-34) an iterative add/subtract loop is executed. In cycle 3 an early-in detection is performed in parallel with the add/subtract. The adjusted  $rs$  operand is detected to be zero extended on the upper most 8, 16 or 24 bits. If this is the case the following 7, 15 or 23 cycles of the add/subtract iterations are skipped.

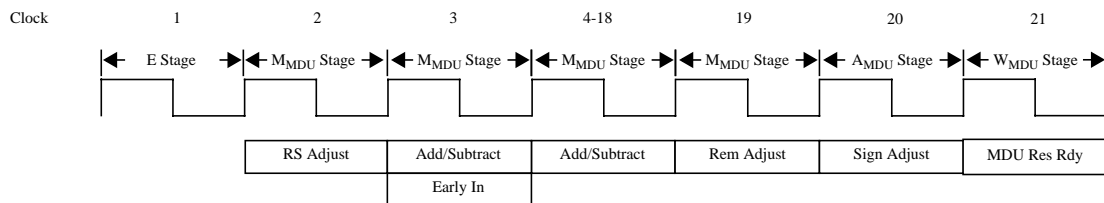
The remainder adjust (Rem Adjust) cycle is required if the remainder was negative. Note that this cycle is spent even if the remainder was positive. A sign adjust is performed on the quotient and/or remainder if necessary. The sign adjust stage is skipped if both operands are positive. In this case the Rem Adjust is moved to the  $A_{MDU}$  stage.

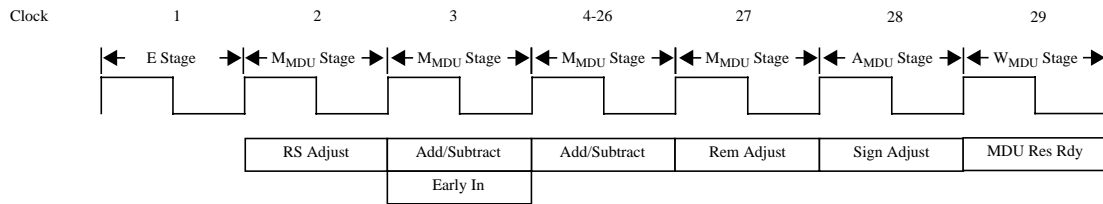
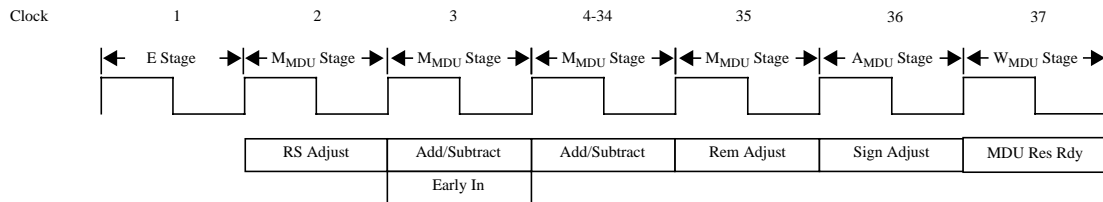
Figure 2.6, Figure 2.7, Figure 2.8 and Figure 2.9 show the latency for 8, 16, 24 and 32 bit divide operations, respectively. The repeat rate is either 11, 19, 27 or 35 cycles (one less if the *sign adjust* stage is skipped) as a second divide can be in the *RS Adjust* stage when the first divide is in the *Reg WR* stage.

**Figure 2.6 High-Performance MDU Pipeline Flow During a 8-bit Divide (DIV) Operation**



**Figure 2.7 High-Performance MDU Pipeline Flow During a 16-bit Divide (DIV) Operation**



**Figure 2.8 High-Performance MDU Pipeline Flow During a 24-bit Divide (DIV) Operation****Figure 2.9 High-Performance MDU Pipeline Flow During a 32-bit Divide (DIV) Operation**

## 2.4 MDU Pipeline (Area-Efficient MDU)

The area-efficient multiply/divide unit (MDU) is a separate autonomous block for multiply and divide operations. The MDU is not pipelined, but rather performs the computations iteratively in parallel with the integer unit (IU) pipeline and does not stall when the IU pipeline stalls. This allows the long-running MDU operations to be partially masked by system stalls and/or other integer unit instructions.

The MDU consists of one 32-bit adder result-accumulate registers (HI and LO), a combined multiply/divide state machine, and all multiplexers and control logic. A simple 1-bit-per-clock recursive algorithm is used for both multiply and divide operations. Using Booth's algorithm all multiply operations complete in 32 clocks. Two extra clocks are needed for multiply-accumulate. The non-restoring algorithm used for divide operations will not work with negative numbers. Adjustment before and after are thus required depending on the sign of the operands. All divide operations complete in 33 to 35 clocks.

Table 2.3 lists the latencies (number of cycles until a result is available) for multiply and divide instructions. The latencies are listed in terms of pipeline clocks. In this table 'latency' refers to the number of cycles necessary for the second instruction to use the results of the first.

**Table 2.3 M14K™ Core Instruction Latencies (Area-Efficient MDU)**

Operand Signs of 1st Instruction (Rs,Rt)	Instruction Sequence		Latency Clocks
	1st Instruction	2nd Instruction	
any, any	MULT/MULTU	MADD/MADDU, MSUB/MSUBU, or MFHI/MFLO	32
any, any	MADD/MADDU, MSUB/MSUBU	MADD/MADDU, MSUB/MSUBU, or MFHI/MFLO	34
any, any	MUL	Integer operation <sup>[1]</sup>	32
any, any	DIVU	MFHI/MFLO	33

**Table 2.3 M14K™ Core Instruction Latencies (Area-Efficient MDU)**

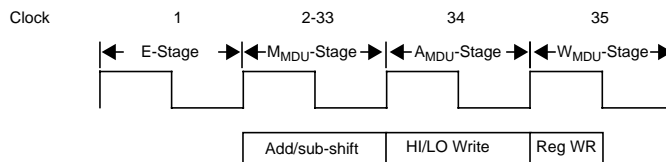
Operand Signs of 1st Instruction (Rs,Rt)	Instruction Sequence		Latency Clocks
	1st Instruction	2nd Instruction	
pos, pos	DIV	MFHI/MFLO	33
any, neg	DIV	MFHI/MFLO	34
neg, pos	DIV	MFHI/MFLO	35
any, any	MFHI/MFLO	Integer operation <sup>[1]</sup>	2
any, any	MTHI/MTLO	MADD/MADDU, MSUB/MSUBU	1

[1] Integer Operation refers to any integer instruction that uses the result of a previous MDU operation.

### 2.4.1 Multiply (Area-Efficient MDU)

Multiply operations are executed using a simple iterative multiply algorithm. Using Booth's approach, this algorithm works for both positive and negative operands. The operation uses 32 cycles in  $M_{MDU}$  stage to complete a multiplication. The register writeback to HI and LO are done in the A stage. For MUL operations, the register file writeback is done in the  $W_{MDU}$  stage.

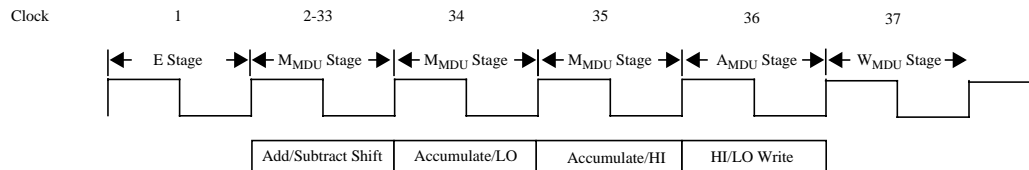
Figure 2.10 shows the latency for a multiply operation. The repeat rate is 33 cycles as a second multiply can be in the first  $M_{MDU}$  stage when the first multiply is in  $A_{MDU}$  stage.

**Figure 2.10 M14K™ Area-Efficient MDU Pipeline Flow During a Multiply Operation**

### 2.4.2 Multiply Accumulate (Area-Efficient MDU)

Multiply-accumulate operations use the same multiply machine as used for multiply only. Two extra stages are needed to perform the addition/subtraction. The operations uses 34 cycles in  $M_{MDU}$  stage to complete the multiply-accumulate. The register writeback to HI and LO are done in the A stage.

Figure 2.11 shows the latency for a multiply-accumulate operation. The repeat rate is 35 cycles as a second multiply-accumulate can be in the E stage when the first multiply is in the last  $M_{MDU}$  stage.

**Figure 2.11 M14K Area-Efficient MDU Pipeline Flow During a Multiply Accumulate Operation**



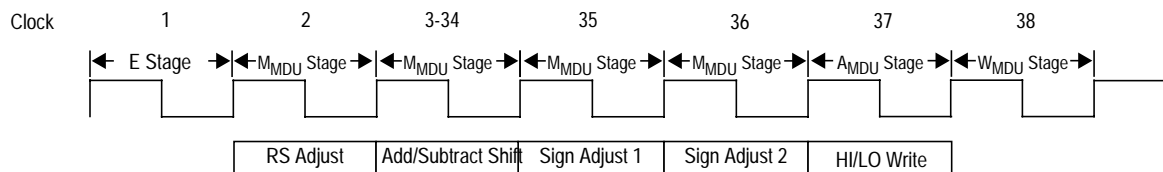
### 2.4.3 Divide (Area-Efficient MDU)

Divide operations also implement a simple non-restoring algorithm. This algorithm works only for positive operands, hence the first cycle of the  $M_{MDU}$  stage is used to negate the  $rs$  operand (RS Adjust) if needed. Note that this cycle is executed even if negation is not needed. The next 32 cycle (3-34) executes an interactive add/subtract-shift function.

Two sign adjust (Sign Adjust 1/2) cycles are used to change the sign of one or both the quotient and the remainder. Note that one or both of these cycles are skipped if they are not needed. The rule is, if both operands were positive or if this is an unsigned division; both of the sign adjust cycles are skipped. If the  $rs$  operand was negative, one of the sign adjust cycles is skipped. If only the  $rs$  operand was negative, none of the sign adjust cycles are skipped. Register writeback to HI and LO are done in the A stage.

Figure 2.12 shows the pipeline flow for a divide operation. The repeat rate is either 34, 35 or 36 cycles (depending on how many sign adjust cycles are skipped) as a second divide can be in the E stage when the first divide is in the last  $M_{MDU}$  stage.

**Figure 2.12 M14K™ Area-Efficient MDU Pipeline Flow During a Divide (DIV) Operation**



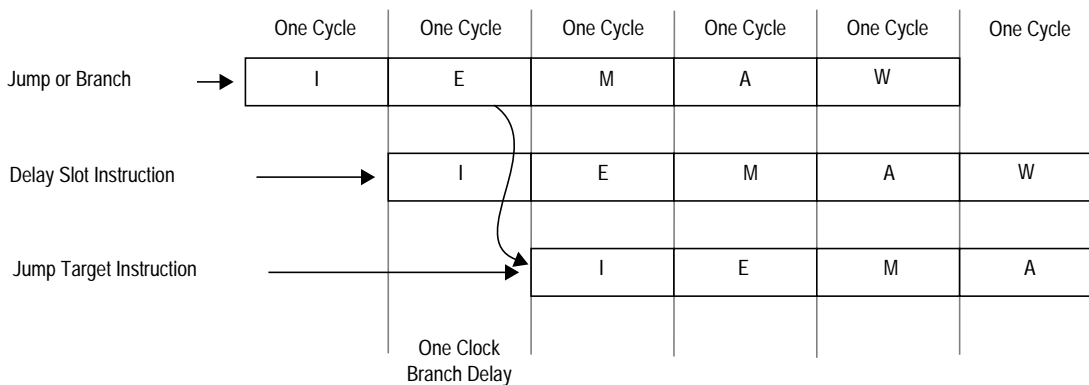
## 2.5 Branch Delay

The pipeline has a branch delay of one cycle. The one-cycle branch delay is a result of the branch decision logic operating during the E pipeline stage. This allows the branch target address to be used in the I stage of the instruction following 2 cycles after the branch instruction. By executing the 1st instruction following the branch instruction sequentially before switching to the branch target, the intervening branch delay slot is utilized. This avoids bubbles being injected into the pipeline on branch instructions. Both the address calculation and the branch condition check are performed in the E stage.

The pipeline begins the fetch of either the branch path or the fall-through path in the cycle following the delay slot. After the branch decision is made, the processor continues with the fetch of either the branch path (for a taken branch) or the fall-through path (for the non-taken branch).

The branch delay means that the instruction immediately following a branch is always executed, regardless of the branch direction. If no useful instruction can be placed after the branch, then the compiler or assembler must insert a NOP instruction in the delay slot.

Figure 2.13 illustrates the branch delay.

**Figure 2.13 IU Pipeline Branch Delay**

## 2.6 Data Bypassing

Most MIPS32 instructions use one or two register values as source operands. These operands are fetched from the register file in the first part of E stage. The ALU straddles the E-to-M boundary, and can present the result early in the M stage. However, the result is not written to the register file before the W stage. If no precautions were taken, it would take 3 cycles before the result was available for the following instructions. To avoid this, data bypassing is implemented.

Between the register file and the ALU a data-bypass multiplexer is placed on both operands (see figure below). This enables the M14K core to forward data from a preceding instruction whose target is a source register of a following instruction. An M to E bypass and an A to E bypass feed the bypass multiplexers. A W to E bypass is not needed, as the register file is capable of making an internal bypass of Rd write data directly to the Rs and Rt read ports.

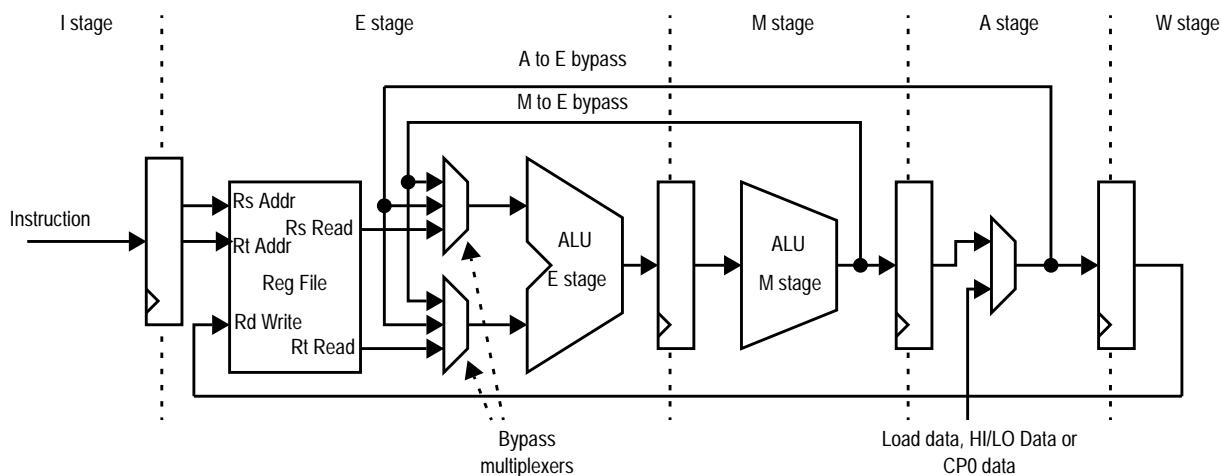
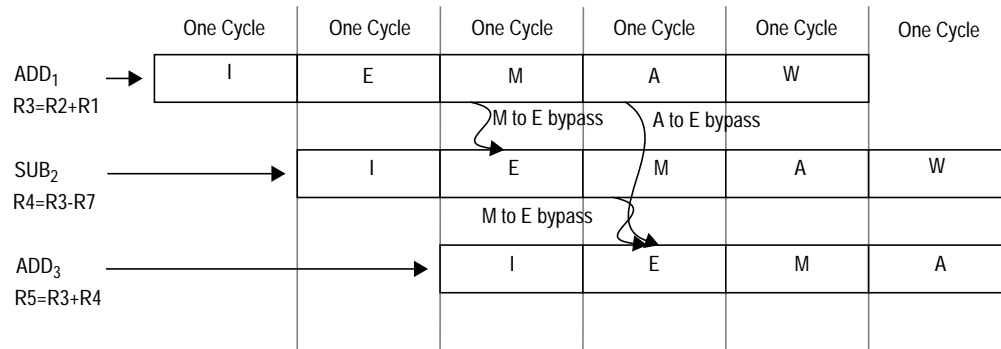
**Figure 2.14 IU Pipeline Data Bypass**

Figure 2.15 shows the data bypass for an  $\text{Add}_1$  instruction followed by a  $\text{Sub}_2$  and another  $\text{Add}_3$  instruction. The  $\text{Sub}_2$  instruction uses the output from the  $\text{Add}_1$  instruction as one of the operands, and thus the M to E bypass is used. The following  $\text{Add}_3$  uses the result from both the first  $\text{Add}_1$  instruction and the  $\text{Sub}_2$  instruction. Since the  $\text{Add}_1$  data is now in A stage, the A to E bypass is used, and the M to E bypass is used to bypass the  $\text{Sub}_2$  data to the  $\text{Add}_2$  instruction.

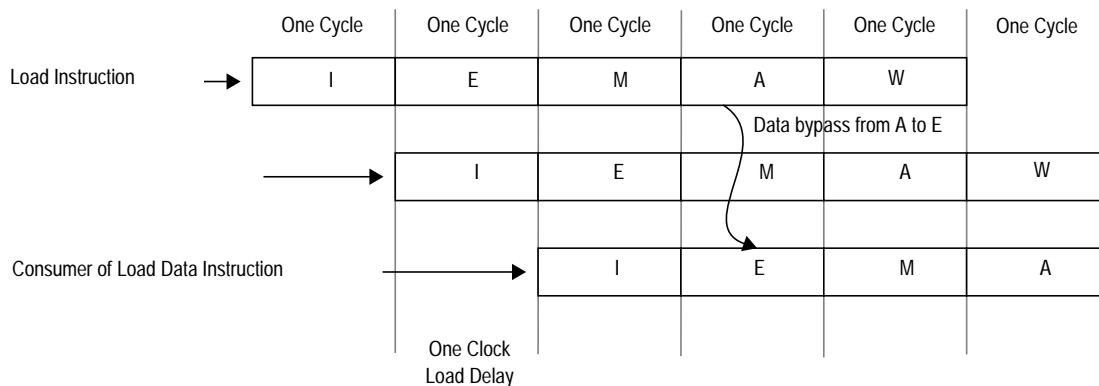
**Figure 2.15 IU Pipeline M to E bypass**



### 2.6.1 Load Delay

Load delay refers to the fact that data fetched by a load instruction is not available in the integer pipeline until after the load aligner in A stage. All instructions need the source operands available in the E stage. An instruction immediately following a load instruction will, if it has the same source register as was the target of the load, cause an instruction interlock pipeline slip in the E stage (see 2.10 “Instruction Interlocks” on page 46). If an instruction following the load by 1 or 2 cycles uses the data from the load, the A to E bypass (see Figure 2.30) serves to reduce or avoid stall cycles. An instruction flow of this is shown in Figure 2.16.

**Figure 2.16 IU Pipeline A to E Data bypass**

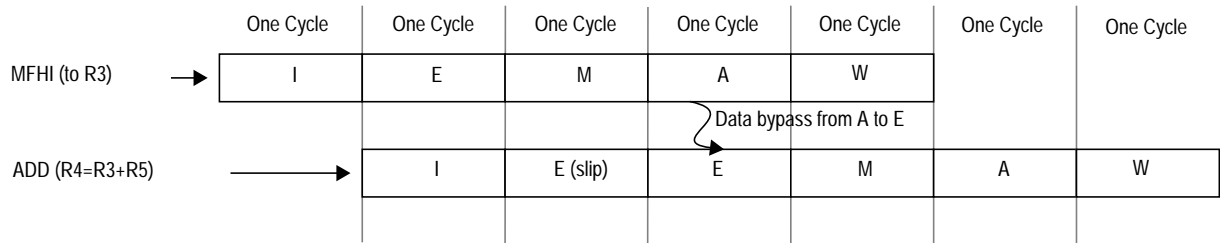


### 2.6.2 Move from HI/LO and CP0 Delay

As indicated in Figure 2.30, not only load data, but also data moved from the HI or LO registers (MFHI/MFLO) and data moved from CP0 (MFC0) enters the IU-Pipeline in the A stage. That is, data is not available in the integer pipeline until early in the A stage. The A to E bypass is available for this data. But as for Loads, an instruction following immediately after one of these move instructions must be paused for one cycle if the target of the move is among the

sources of the following instruction and this causes an interlock slip in the E stage (see 2.10 “Instruction Interlocks” on page 46). An interlock slip after a MFHI is illustrated in Figure 2.17.

**Figure 2.17 IU Pipeline Slip after a MFHI**

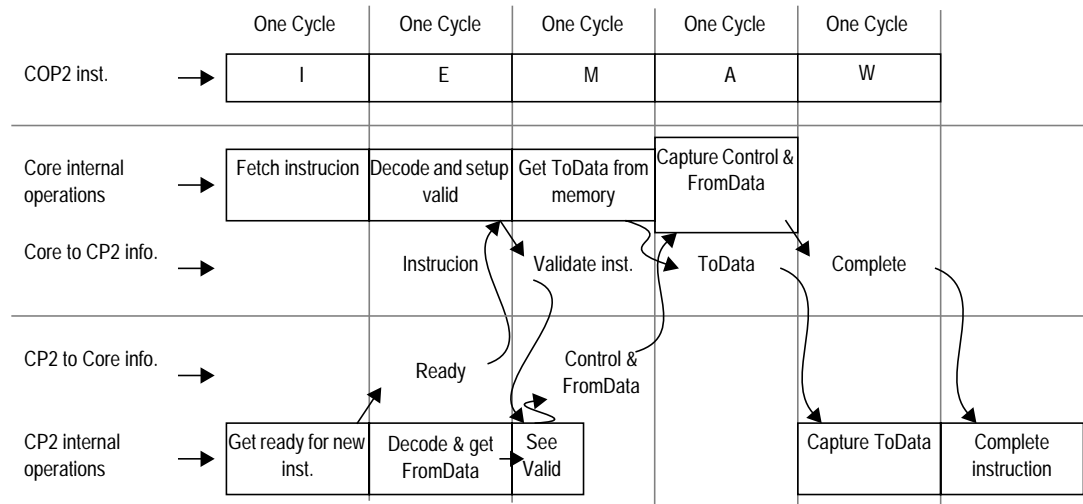


## 2.7 Coprocessor 2 Instructions

If a coprocessor 2 is attached to the M14K core, a number of transactions must take place on the CP2 Interface for each coprocessor 2 instruction. First, if the CU[2] bit in the CP0 *Status* register is not set, then no coprocessor 2 related instruction will start a transaction on the CP2 Interface; instead, a Coprocessor Unusable exception will be signaled. If the CU[2] bit is set, and a coprocessor 2 instruction is fetched, the following transactions will occur on the CP2 Interface:

1. The Instruction is presented on the instructions bus in E stage. Coprocessor 2 can do a decode in the same cycle.
2. The Instruction is validated from the core in M stage. From this point, the core will accept control and data signals back from coprocessor 2. All control and data signals from coprocessor 2 are captured on input latches to the core.
3. If all the expected control and data signals were presented to the core in the previous M stage, the core will proceed to execute the A stage. If some return information is missing, the A stage will not advance and cause a slip in all I, E, and M stages (see 2.9 “Slip Conditions” on page 46).  
If this instruction sent data from the core to coprocessor 2, this data is sent in the A stage.
4. The instruction completion is signaled to coprocessor 2 in the W stage. Potential data from the coprocessor is written to the register file.

Figure 2.18 shows the timing relationship between the M14K core and coprocessor 2 for all coprocessor 2 instructions.

**Figure 2.18 Coprocessor 2 Interface Transactions**

As can be seen in the Figure, all control and data from the coprocessor must occur in the M stage. If this is not the case, the A stage will start slipping in the following cycle and thus stall the I, E, M, and A stages; but if all expected control and data is available in the M stage, coprocessor 2 instructions can execute with no pipeline stalls. The only exception to this is the Branch on Coprocessor conditions (BC2) instruction. All branch instructions, including the regular BEQ, BNE, etc., must be resolved in the E stage. The M14K core does not have branch prediction logic, and thus the target address must be available before the end of the E stage. The BC2 instruction has to follow the same protocol as all other coprocessor 2 instructions on the CP2 Interface. All core interface operations belonging to the E, M, and A stages will have to occur in the E stage for BC2 instructions. This means that a BC2 instruction always slips for a minimum of 2 cycles into the E stage, and any delay in the return of branch information from coprocessor 2 will add to the number of slip cycles. All other Coprocessor 2 instructions can operate without slips, provided that all control and data information from coprocessor 2 is transferred in the M stage.

## 2.8 Interlock Handling

Smooth pipeline flow is interrupted when cache misses occur or when data dependencies are detected. Interruptions handled entirely in hardware, such as cache misses, are referred to as *interlocks*. At each cycle, interlock conditions are checked for all active instructions.

Table 2.4 lists the types of pipeline interlocks for the M14K processor core.

**Table 2.4 Pipeline Interlocks**

Interlock Type	Sources	Slip Stage
I-side SRAM Stall	SRAM Access not complete	E Stage
Instruction	Producer-consumer hazards	E/M Stage
	Hardware Dependencies (MDU)	E Stage
	BC2 waiting for COP2 Condition Check	E Stage
D-side SRAM Stall	SRAM Access not complete	A Stage
Coprocessor 2 completion slip	Coprocessor 2 control and/or data delay from coprocessor	A Stage

In general, MIPS processors support two types of hardware interlocks:

- Stalls, which are resolved by halting the pipeline
- Slips, which allow one part of the pipeline to advance while another part of the pipeline is held static

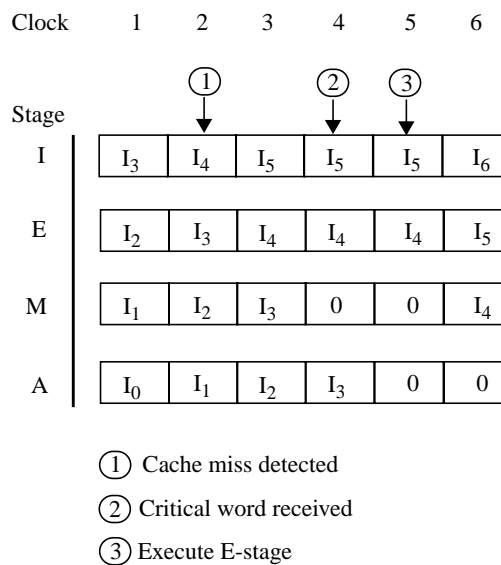
In the M14K processor core, all interlocks are handled as slips.

## 2.9 Slip Conditions

On every clock, internal logic determines whether each pipe stage is allowed to advance. These slip conditions propagate backwards down the pipe. For example, if the M stage does not advance, neither does the E or I stage.

Slipped instructions are retried on subsequent cycles until they issue. The back end of the pipeline advances normally during slips. This resolves the conflict when the slip was caused by a missing result. NOPs are inserted into the bubble in the pipeline. [Figure 2.19](#) shows an instruction cache miss that causes a two-cycle slip.

**Figure 2.19 Instruction Cache Miss Slip**



In the first clock cycle in [Figure 2.19](#), the pipeline is full and the cache miss is detected. Instruction I<sub>0</sub> is in the A stage, instruction I<sub>1</sub> is in the M stage, instruction I<sub>2</sub> is in the E stage, and instruction I<sub>3</sub> is in the I stage. The cache miss occurs in clock 2 when the I<sub>4</sub> instruction fetch is attempted. I<sub>4</sub> advances to the E stage and waits for the instruction to be fetched from main memory. In this example, two clocks (3 and 4) are required to fetch the I<sub>4</sub> instruction from memory. After the cache miss has been resolved in clock 4 and the instruction is bypassed to the E stage, the pipeline is restarted, causing I<sub>4</sub> to finally execute its E-stage operations.

## 2.10 Instruction Interlocks

Most instructions can be issued at a rate of one per clock cycle. In order to adhere to the sequential programming model, the issue of an instruction must sometimes be delayed to ensure that the result of a prior instruction is avail-

able. Table 2.5 details the instruction interactions that prevent an instruction from advancing in the processor pipeline.

**Table 2.5 Instruction Interlocks**

Instruction Interlocks				
First Instruction		Second Instruction	Issue Delay (in Clock Cycles)	Slip Stage
LB/LBU/LH/LHU/LL/LW/LWL/LWR		Consumer of load data	1	E stage
MFC0		Consumer of destination register	1	E stage
MULTx/MADDx/MSUBx (high-performance MDU)	16bx32b	MFLO/MFHI	0	
	32bx32b		1	M stage
MUL (high-performance MDU)	16bx32b	Consumer of target data	2	E stage
	32bx32b		3	E stage
MUL (high-performance MDU)	16bx32b	Non-Consumer of target data	1	E stage
	32bx32b		2	E stage
MFHI/MFLO		Consumer of target data	1	E stage
MULTx/MADDx/MSUBx (high-performance MDU)	16bx32b	MULT/MUL/MADD/MSUB MTHI/MTLO/DIV	0 <sup>[1]</sup>	E stage
	32bx32b		1 <sup>[1]</sup>	E stage
DIV		MUL/MULTx/MADDx/ MSUBx/MTHI/MTLO/ MFHI/MFLO/DIV	Until DIV completes	E stage
MULT/MUL/MADD/MSUB/MTHI/MTLO/MFHI/MFLO/DIV (area-efficient MDU)		MULT/MUL/MADD/MSUB/ MTHI/MTLO/MFHI/MFLO/ DIV	Until 1st MDU op completes	E stage
MUL (area-efficient MDU)		Any Instruction	Until MUL completes	E stage
MFC0/MFC2/CFC2		Consumer of target data	1	E stage

## 2.11 Hazards

In general, the M14K core ensures that instructions are executed following a fully sequential program model in which each instruction in the program sees the results of the previous instruction. There are some deviations to this model, referred to as *hazards*.

Prior to Release 2 of the MIPS32® Architecture, hazards (primarily CP0 hazards) were relegated to implementation-dependent cycle-based solutions, primarily based on the SSNOP instruction. This has been an insufficient and error-prone practice that must be addressed with a firm compact between hardware and software. As such, new instructions have been added to Release 2 of the architecture which act as explicit barriers that eliminate hazards. To the extent that it was possible to do so, the new instructions have been added in such a way that they are backward-compatible with existing MIPS processors.

### 2.11.1 Types of Hazards

With one exception, all hazards were eliminated in Release 1 of the Architecture for unprivileged software. The exception occurs when unprivileged software writes a new instruction sequence and then wishes to jump to it. Such an operation remained a hazard, and is addressed by the capabilities of Release 2.

In privileged software, there are two types of hazards: *execution hazards* and *instruction hazards*.

Execution hazards are those created by the execution of one instruction, and seen by the execution of another instruction. Table 2.6 lists execution hazards.

**Table 2.6 Execution Hazards**

Producer	→	Consumer	Hazard On	Spacing (Instructions)
MTC0	→	Coprocessor instruction execution depends on the new value of Status <sub>CU</sub>	Status <sub>CU</sub>	1
MTC0	→	ERET	EPC DEPC ErrorEPC	1
MTC0	→	ERET	Status	0
MTC0, EI, DI	→	Interrupted Instruction	Status <sub>IE</sub>	1
MTC0	→	Interrupted Instruction	Cause <sub>IP</sub>	3
MTC0	→	RDPGPR WRPGPR	SRSCtl <sub>PSS</sub>	1
MTC0	→	Instruction not seeing a Timer Interrupt	Compare update that clears Timer Interrupt	4 <sup>1</sup>
MTC0	→	Instruction affected by change	Any other CP0 register	2

1. This is the minimum value. Actual value is system-dependent since it is a function of the sequential logic between the *SL\_TimerInt* output and the external logic which feeds *SL\_TimerInt* back into one of the *SL\_Int* inputs, or a function of the method for handling *SL\_TimerInt* in an external interrupt controller.

Instruction hazards are those created by the execution of one instruction, and seen by the instruction fetch of another instruction. Table 2.7 lists instruction hazards.

**Table 2.7 Instruction Hazards**

Producer	→	Consumer	Hazard On	Spacing (Instructions)
MTC0	→	Instruction fetch seeing the new value (including a change to ERL followed by an instruction fetch from the useg segment)	Status	
Instruction stream write via redirected store	→	Instruction fetch seeing the new instruction stream	Cache entries	3



## 2.11.2 Instruction Listing

Table 2.8 lists the instructions designed to eliminate hazards. See the document titled *MIPS32® Architecture for Programmers Volume II: The MIPS32® Instruction Set* (MD00086) for a more detailed description of these instructions.

**Table 2.8 Hazard Instruction Listing**

Mnemonic	Function
EHB	Clear execution hazard
JALR.HB	Clear both execution and instruction hazards
JR.HB	Clear both execution and instruction hazards
SYNCI	Synchronize caches after instruction stream write

### 2.11.2.1 Instruction Encoding

The EHB instruction is encoded using a variant of the NOP/SSNOP encoding. This encoding was chosen for compatibility with the Release 1 SSNOP instruction, such that existing software may be modified to be compatible with both Release 1 and Release 2 implementations. See the EHB instruction description for additional information.

The JALR.HB and JR.HB instructions are encoding using bit 10 of the *hint* field of the JALR and JR instructions. These encodings were chosen for compatibility with existing MIPS implementations, including many which pre-date the MIPS32 architecture. Because a pipeline flush clears hazards on most early implementations, the JALR.HB or JR.HB instructions can be included in existing software for backward and forward compatibility. See the JALR.HB and JR.HB instructions for additional information.

The SYNCI instruction is encoded using a new encoding of the REGIMM opcode. This encoding was chosen because it causes a Reserved Instruction exception on all Release 1 implementations. As such, kernel software running on processors that don't implement Release 2 can emulate the function using the CACHE instruction.

## 2.11.3 Eliminating Hazards

The Spacing column shown in Table 2.6 and Table 2.7 indicates the number of unrelated instructions (such as NOPs or SSNOPs) that, prior to the capabilities of Release 2, would need to be placed between the producer and consumer of the hazard in order to ensure that the effects of the first instruction are seen by the second instruction. Entries in the table that are listed as 0 are traditional MIPS hazards which are not hazards on the M14K core.

With the hazard elimination instructions available in Release 2, the preferred method to eliminate hazards is to place one of the instructions listed in Table 2.8 between the producer and consumer of the hazard. Execution hazards can be removed by using the EHB, JALR.HB, or JR.HB instructions. Instruction hazards can be removed by using the JALR.HB or JR.HB instructions, in conjunction with the SYNCI instruction. Since the M14K core does not contain caches, the SYNCI instruction is not strictly necessary, but is still recommended to create portable code that can be run on other MIPS processors that may contain caches.



## Memory Management of the M14K™ Core

The M14K processor core includes a Memory Management Unit (MMU) that interfaces between the execution unit and the cache controller. The core implements a simple Fixed Mapping (FM) style MMU.

This chapter contains the following sections:

- [Section 3.1 “Introduction”](#)
- [Section 3.2 “Modes of Operation”](#)
- [Section 3.3 “Fixed Mapping MMU”](#)
- [Section 3.4 “System Control Coprocessor”](#)

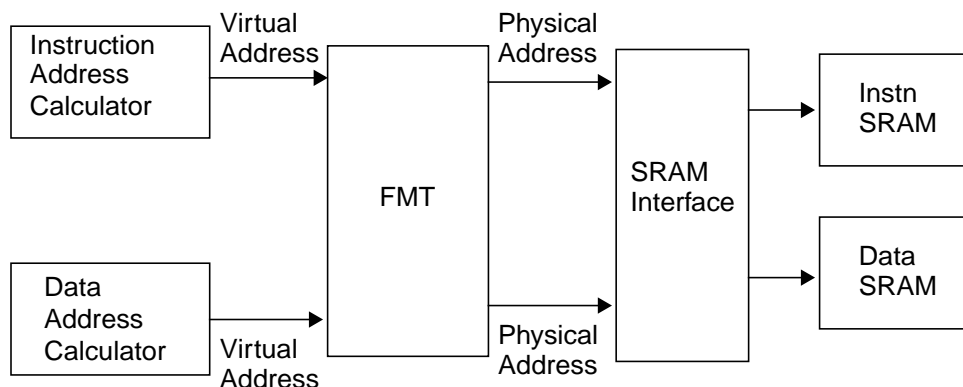
### 3.1 Introduction

The MMU will translate any virtual address to a physical address before a request is sent to the SRAM interface for an external memory reference.

In the M14K processor core, the MMU is based on a simple algorithm to translate virtual addresses into physical addresses via a Fixed Mapping (FM) mechanism. These translations are different for various regions of the virtual address space (useg/kuseg, kseg0, kseg1, kseg2/3).

[Figure 3.1](#) shows how the memory management unit interacts with the SRAM access in the M14K core.

**Figure 3.1 Address Translation During SRAM Access**



### 3.2 Modes of Operation

An M14K processor core supports three modes of operation:

- User mode
- Kernel mode
- Debug mode

User mode is most often used for application programs. Kernel mode is typically used for handling exceptions and privileged operating system functions, including CP0 management and I/O device accesses. Debug mode is used for software debugging and most likely occurs within a software development tool.

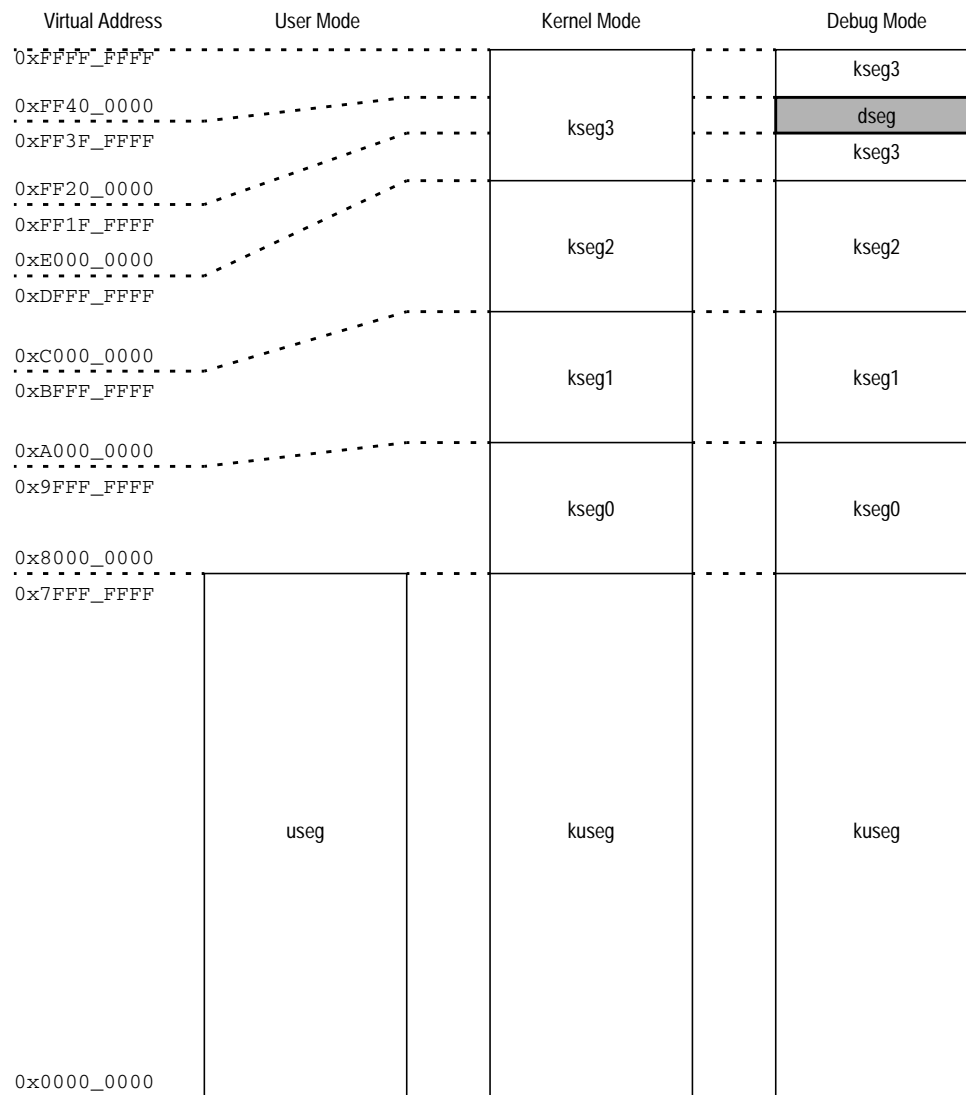
The address translation performed by the MMU depends on the mode in which the processor is operating.

### 3.2.1 Virtual Memory Segments

The Virtual memory segments are different depending on the mode of operation. [Figure 3.2](#) shows the segmentation for the 4 GByte ( $2^{32}$  bytes) virtual memory space addressed by a 32-bit virtual address, for the three modes of operation.

The core enters Kernel mode both at reset and when an exception is recognized. While in Kernel mode, software has access to the entire address space, as well as all CP0 registers. User mode accesses are limited to a subset of the virtual address space (0x0000\_0000 to 0x7FFF\_FFFF) and can be inhibited from accessing CP0 functions. In User mode, virtual addresses 0x8000\_0000 to 0xFFFF\_FFFF are invalid and cause an exception if accessed.

Debug mode is entered on a debug exception. While in Debug mode, the debug software has access to the same address space and CP0 registers as for Kernel mode. In addition, while in Debug mode the core has access to the debug segment dseg. This area overlays part of the kernel segment kseg3. dseg access in Debug mode can be turned on or off, allowing full access to the entire kseg3 in Debug mode, if so desired.

**Figure 3.2 M14K™ processor core Virtual Memory Map**

Each of the segments shown in [Figure 3.2](#) are either mapped or unmapped. The following two sub-sections explain the distinction. Then sections [3.2.2 “User Mode”](#), [3.2.3 “Kernel Mode”](#) and [3.2.4 “Debug Mode”](#) specify which segments are actually mapped and unmapped.

### 3.2.1.1 Unmapped Segments

An unmapped segment does not use the FM to translate from virtual-to-physical addresses.

Unmapped segments have a fixed simple translation from virtual to physical address. This is much like the translations the FM provides for the M14K core, but we will still make the distinction.

All segments are treated as uncached within the M14K core. Cache coherency attributes of cached or uncached can be specified and this information will be sent with the request to allow the system to make a distinction between the two.

### 3.2.1.2 Mapped Segments

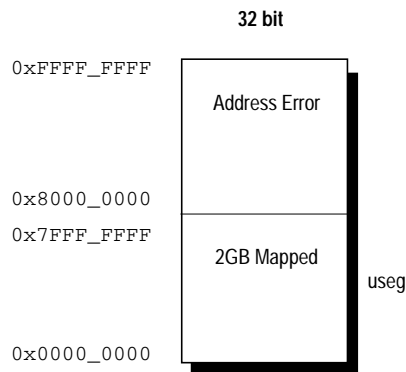
A mapped segment does use the FM to translate from virtual-to-physical addresses.

For the M14K core, the mapped segments have a fixed translation from virtual to physical address. The cacheability of the segment is defined in the CP0 *Config* register fields *K23* and *KU* (see 5.2.20 “Config Register (CP0 Register 16, Select 0)”). Write protection of segments is not possible during FM translation.

### 3.2.2 User Mode

In user mode, a single 2 GByte ( $2^{31}$  bytes) uniform virtual address space called the user segment (useg) is available. Figure 3.3 shows the location of user mode virtual address space.

**Figure 3.3 User Mode Virtual Address Space**



The user segment starts at address 0x0000\_0000 and ends at address 0x7FFF\_FFFF. Accesses to all other addresses cause an address error exception.

The processor operates in User mode when the *Status* register contains the following bit values:

- $UM = 1$
- $EXL = 0$
- $ERL = 0$

In addition to the above values, the DM bit in the *Debug* register must be 0.

Table 3.1 lists the characteristics of the useg User mode segments.

**Table 3.1 User Mode Segments**

Address Bit Value	Status Register			Segment Name	Address Range	Segment Size
	Bit Value					
	EXL	ERL	UM			
32-bit A(31) = 0	0	0	1	useg	0x0000_0000 --> 0x7FFF_FFFF	2 GByte (2 <sup>31</sup> bytes)

All valid user mode virtual addresses have their most significant bit cleared to 0, indicating that user mode can only access the lower half of the virtual memory map. Any attempt to reference an address with the most significant bit set while in user mode causes an address error exception.

The system maps all references to *useg* through the FMT.

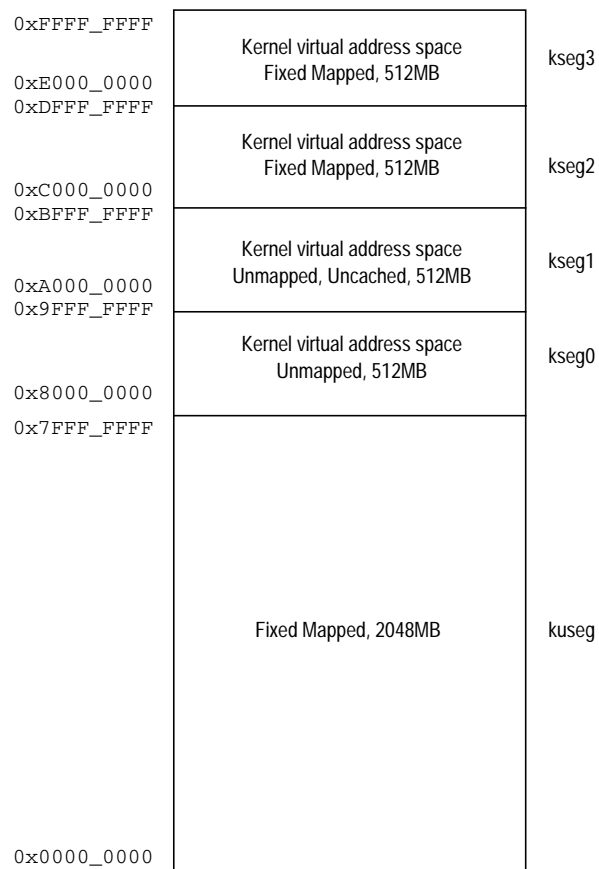
### 3.2.3 Kernel Mode

The processor operates in Kernel mode when the *DM* bit in the *Debug* register is 0 and the *Status* register contains one or more of the following values:

- *UM* = 0
- *ERL* = 1
- *EXL* = 1

When a non-debug exception is detected, *EXL* or *ERL* will be set and the processor will enter Kernel mode. At the end of the exception handler routine, an Exception Return (ERET) instruction is generally executed. The ERET instruction jumps to the Exception PC, clears *ERL*, and clears *EXL* if *ERL*=0. This may return the processor to User mode.

Kernel mode virtual address space is divided into regions differentiated by the high-order bits of the virtual address, as shown in [Figure 3.4](#). Also, [Table 3.2](#) lists the characteristics of the Kernel mode segments.

**Figure 3.4 Kernel Mode Virtual Address Space****Table 3.2 Kernel Mode Segments**

Address Bit Values	Status Register Is One of These Values			Segment Name	Address Range	Segment Size
	UM	EXL	ERL			
A(31) = 0	(UM = 0 or EXL = 1 or ERL = 1) and DM = 0			kuseg	0x0000_0000 through 0x7FFF_FFFF	2 GBytes (2 <sup>31</sup> bytes)
A(31:29) = 100 <sub>2</sub>				kseg0	0x8000_0000 through 0x9FFF_FFFF	512 MBytes (2 <sup>29</sup> bytes)
A(31:29) = 101 <sub>2</sub>				kseg1	0xA000_0000 through 0xBFFF_FFFF	512 MBytes (2 <sup>29</sup> bytes)
A(31:29) = 110 <sub>2</sub>				kseg2	0xC000_0000 through 0xDFFF_FFFF	512 MBytes (2 <sup>29</sup> bytes)
A(31:29) = 111 <sub>2</sub>				kseg3	0xE000_0000 through 0xFFFF_FFFF	512 MBytes (2 <sup>29</sup> bytes)



### 3.2.3.1 Kernel Mode, User Space (kuseg)

In Kernel mode, when the most-significant bit of the virtual address (A31) is cleared, the 32-bit kuseg virtual address space is selected and covers the full  $2^{31}$  bytes (2 GBytes) of the current user address space mapped to addresses 0x0000\_0000 - 0x7FFF\_FFFF.

When  $ERL = 1$  in the *Status* register, the user address region becomes a  $2^{31}$ -byte unmapped and uncached address space. While in this setting, the kuseg virtual address maps directly to the same physical address.

### 3.2.3.2 Kernel Mode, Kernel Space 0 (kseg0)

In Kernel mode, when the most-significant three bits of the virtual address are  $100_2$ , 32-bit kseg0 virtual address space is selected; it is the  $2^{29}$ -byte (512-MByte) kernel virtual space located at addresses 0x8000\_0000 - 0x9FFF\_FFFF. References to kseg0 are unmapped; the physical address selected is defined by subtracting 0x8000\_0000 from the virtual address. The *K0* field of the *Config* register controls cacheability.

### 3.2.3.3 Kernel Mode, Kernel Space 1 (kseg1)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are  $101_2$ , 32-bit kseg1 virtual address space is selected. kseg1 is the  $2^{29}$ -byte (512-MByte) kernel virtual space located at addresses 0xA000\_0000 - 0xBFFF\_FFFF. References to kseg1 are unmapped; the physical address selected is defined by subtracting 0xA000\_0000 from the virtual address.

### 3.2.3.4 Kernel Mode, Kernel Space 2 (kseg2)

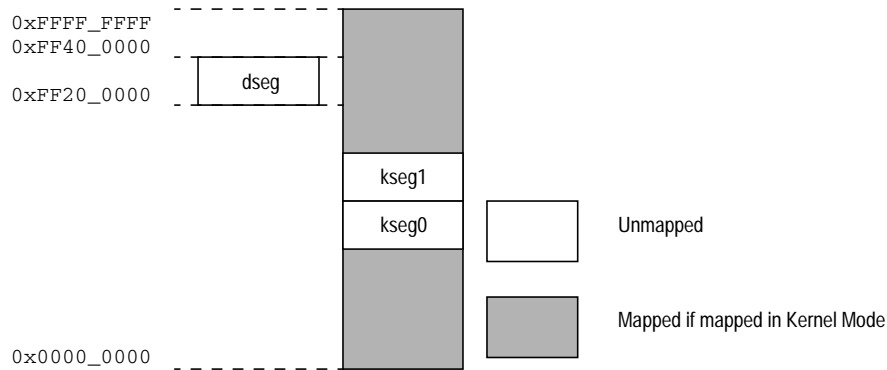
In Kernel mode, when  $UM = 0$ ,  $ERL = 1$ , or  $EXL = 1$  in the *Status* register, and  $DM = 0$  in the *Debug* register, and the most-significant three bits of the 32-bit virtual address are  $110_2$ , 32-bit kseg2 virtual address space is selected. In the M14K core, this  $2^{29}$ -byte (512-MByte) kernel virtual space is located at physical addresses 0xC000\_0000 - 0xDFFF\_FFFF.

### 3.2.3.5 Kernel Mode, Kernel Space 3 (kseg3)

In Kernel mode, when the most-significant three bits of the 32-bit virtual address are  $111_2$ , the kseg3 virtual address space is selected. In the M14K core, this  $2^{29}$ -byte (512-MByte) kernel virtual space is located at physical addresses 0xE000\_0000 - 0xFFFF\_FFFF.

## 3.2.4 Debug Mode

Debug mode address space is identical to Kernel mode address space with respect to mapped and unmapped areas, except for *kseg3*. In *kseg3*, a debug segment *dseg* co-exists in the virtual address range 0xFF20\_0000 to 0xFF3F\_FFFF. The layout is shown in [Figure 3.5](#).

**Figure 3.5 Debug Mode Virtual Address Space**

The dseg is sub-divided into the dmseg segment at 0xFF20\_0000 to 0xFF2F\_FFFF which is used when the probe services the memory segment, and the drseg segment at 0xFF30\_0000 to 0xFF3F\_FFFF which is used when memory-mapped debug registers are accessed. The subdivision and attributes for the segments are shown in [Table 3.3](#).

Accesses to memory that would normally cause an exception if tried from kernel mode cause the core to re-enter debug mode via a debug mode exception.

The unmapped kseg0 and kseg1 segments from kernel mode address space are available from debug mode, which allows the debug handler to be executed from uncached and unmapped memory.

**Table 3.3 Physical Address and Cache Attributes for dseg, dmseg, and drseg Address Spaces**

Segment Name	Sub-Segment Name	Virtual Address	Generates Physical Address	Cache Attribute
dseg	dmseg	0xFF20_0000 through 0xFF2F_FFFF	dmseg maps to addresses 0x0_0000 - 0xF_FFFF in EJTAG probe memory space.	Uncached
	drseg	0xFF30_0000 through 0xFF3F_FFFF	drseg maps to the breakpoint registers 0x0_0000 - 0xF_FFFF	

#### 3.2.4.1 Conditions and Behavior for Access to drseg, EJTAG Registers

The behavior of CPU access to the drseg address range at 0xFF30\_0000 to 0xFF3F\_FFFF is determined as shown in [Table 3.4](#)

**Table 3.4 CPU Access to drseg Address Range**

Transaction	LSNM Bit in Debug Register	Access
Load / Store	1	Kernel mode address space (kseg3)
Fetch	Don't care	drseg, see comments below
Load / Store	0	

Debug software is expected to read the Debug Control Register (*DCR*) to determine which other memory mapped registers exist in drseg. The value returned in response to a read of any unimplemented memory mapped register is

unpredictable, and writes are ignored to any unimplemented register in the drseg. Refer to [Chapter 8, “EJTAG Debug Support in the M14K™ Core”](#) on page 159 for more information on the *DCR*.

The allowed access size is limited for the drseg. Only word size transactions are allowed. Operation of the processor is undefined for other transaction sizes.

### 3.2.4.2 Conditions and Behavior for Access to dmseg, EJTAG Memory

The behavior of CPU access to the dmseg address range at 0xFF20\_0000 to 0xFF2F\_FFFF is determined by the table shown in [Table 3.5](#).

**Table 3.5 CPU Access to dmseg Address Range**

Transaction	ProbEn bit in DCR register	LSNM bit in Debug register	Access
Load / Store	Don't care	1	Kernel mode address space (kseg3)
Fetch	1	Don't care	dmseg
Load / Store	1	0	
Fetch	0	Don't care	See comments below
Load / Store	0	0	

The case with access to the dmseg when the *ProbEn* bit in the *DCR* register is 0 is not expected to happen. Debug software is expected to check the state of the *ProbEn* bit in *DCR* register before attempting to reference dmseg. If such a reference does happen, the reference hangs until it is satisfied by the probe. The probe can not assume that there will never be a reference to dmseg if the *ProbEn* bit in the *DCR* register is 0 because there is an inherent race between the debug software sampling the *ProbEn* bit as 1 and the probe clearing it to 0.

## 3.3 Fixed Mapping MMU

The M14K core implements a simple Fixed Mapping (FM) memory management unit that is smaller than the a full translation lookaside buffer (TLB) and more easily synthesized. Like a TLB, the FM performs virtual-to-physical address translation and provides attributes for the different memory segments. Those memory segments which are unmapped in a TLB implementation (kseg0 and kseg1) are translated identically by the FM MMU.

The FM also determines the cacheability of each segment. These attributes are controlled via bits in the *Config* register. [Table 3.6](#) shows the encoding for the K23 (bits 30:28), KU (bits 27:25) and K0 (bits 2:0) of the *Config* register.

The M14K core does not contain caches and will treat all references as uncached, but these *Config* fields will be sent out to the system with the request and it can choose to use them to control any external caching that may be present..

**Table 3.6 Cacheability of Segments with Block Address Translation**

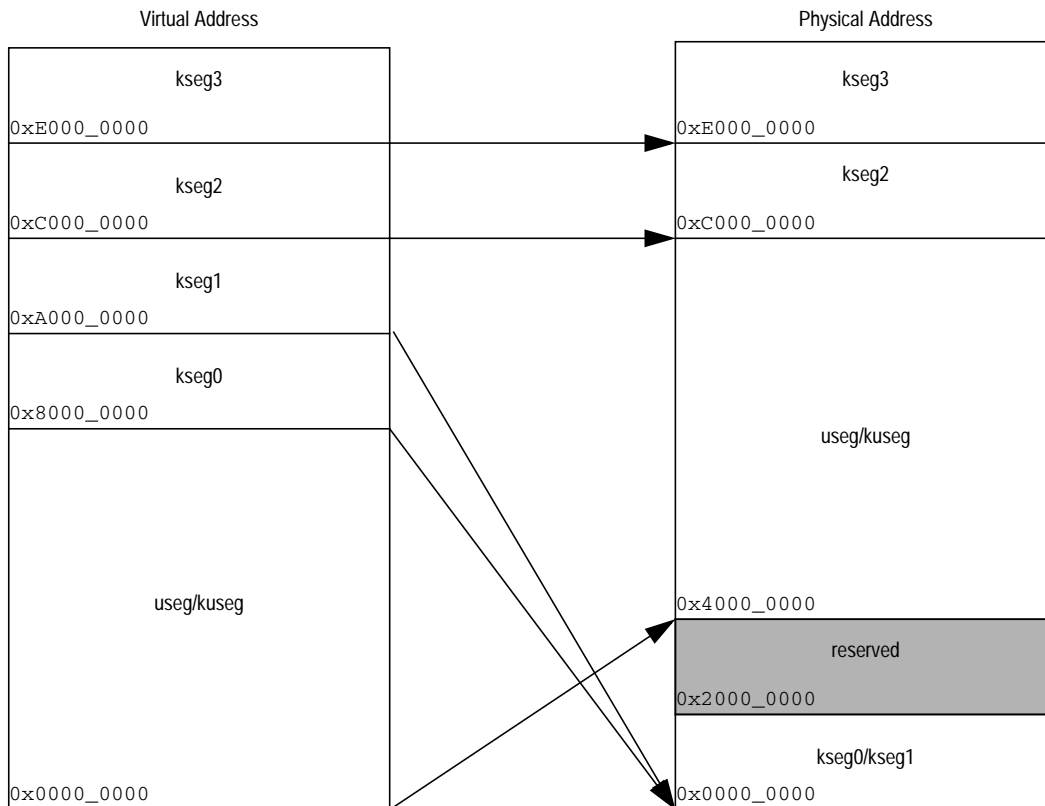
Segment	Virtual Address Range	Cacheability
useg/kuseg	0x0000_0000-0x7FFF_FFFF	Controlled by the KU field (bits 27:25) of the <i>Config</i> register.
kseg0	0x8000_0000-0x9FFF_FFFF	Controlled by the K0 field (bits 2:0) of the <i>Config</i> register.

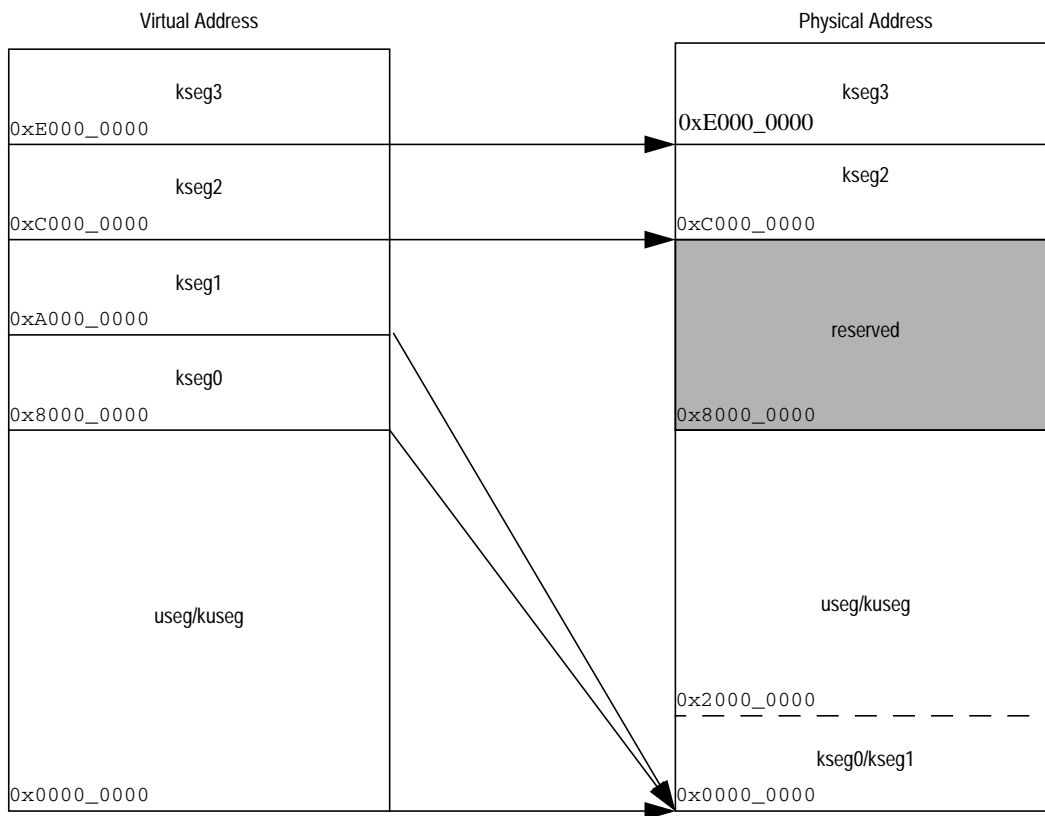
**Table 3.6 Cacheability of Segments with Block Address Translation (Continued)**

Segment	Virtual Address Range	Cacheability
kseg1	0xA000_0000-0xBFFF_FFFF	Always uncacheable.
kseg2	0xC000_0000-0xDFFF_FFFF	Controlled by the K23 field (bits 30:28) of the <i>Config</i> register.
kseg3	0xE000_0000-0xFFFF_FFFF	Controlled by K23 field (bits 30:28) of the <i>Config</i> register.

The FM performs a simple translation to map from virtual addresses to physical addresses. This mapping is shown in [Figure 3.6](#). When ERL=1, useg and kuseg become unmapped and uncached. The ERL behavior is the same as if there was a TLB. The ERL mapping is shown in [Figure 3.7](#).

The ERL bit is usually never asserted by software. It is asserted by hardware after a Reset, SoftReset or NMI. See [4.8 “Exception Descriptions” on page 82](#) for further information on exceptions.

**Figure 3.6 FM Memory Map (ERL=0) in the M14K™ Processor Core**

**Figure 3.7 FM Memory Map (ERL=1) in the M14K™ Processor Core**

### 3.4 System Control Coprocessor

The System Control Coprocessor (CP0) is implemented as an integral part of M14K processor core and supports memory management, address translation, exception handling, and other privileged operations. Certain CP0 registers are used to support memory management. Refer to [Chapter 5, “CP0 Registers of the M14K™ Core”](#) on page 95 for more information on the CP0 register set.



## Exceptions and Interrupts in the M14K™ Core

The M14K processor core receives exceptions from a number of sources, including arithmetic overflows, I/O interrupts, and system calls. When the CPU detects one of these exceptions, the normal sequence of instruction execution is suspended and the processor enters kernel mode.

In kernel mode the core disables interrupts and forces execution of a software exception processor (called a handler) located at a specific address. The handler saves the context of the processor, including the contents of the program counter, the current operating mode, and the status of the interrupts (enabled or disabled). This context is saved so it can be restored when the exception has been serviced.

When an exception occurs, the core loads the *Exception Program Counter (EPC)* register with a location where execution can restart after the exception has been serviced. Most exceptions are *precise*, which mean that *EPC* can be used to identify the instruction that caused the exception. For precise exceptions, the restart location in the *EPC* register is the address of the instruction that caused the exception or, if the instruction was executing in a branch delay slot, the address of the branch instruction immediately preceding the delay slot. To distinguish between the two, software must read the BD bit in the CP0 *Cause* register. Bus error exceptions and CP2 exceptions may be imprecise. For imprecise exceptions the instruction that caused the exception cannot be identified.

This chapter contains the following sections:

- [Section 4.1 “Exception Conditions”](#)
- [Section 4.2 “Exception Priority”](#)
- [Section 4.3 “Interrupts”](#)
- [Section 4.4 “GPR Shadow Registers”](#)
- [Section 4.5 “Exception Vector Locations”](#)
- [Section 4.6 “General Exception Processing”](#)
- [Section 4.7 “Debug Exception Processing”](#)
- [Section 4.8 “Exception Descriptions”](#)
- [Section 4.9 “Exception Handling and Servicing Flowcharts”](#)

### 4.1 Exception Conditions

When an exception condition occurs, the instruction causing the exception and all those that follow it in the pipeline are cancelled (“flushed”). Accordingly, any stall conditions and any later exception conditions that might have referenced this instruction are inhibited—obviously there is no benefit in servicing stalls for a cancelled instruction.

When an exception condition is detected on an instruction fetch, the core aborts that instruction and all instructions that follow. When this instruction reaches the W stage, various CP0 registers are written with the exception state, change the current program counter (PC) to the appropriate exception vector address, and clearing the exception bits of earlier pipeline stages.

This implementation allows all preceding instructions to complete execution and prevents all subsequent instructions from completing. Thus, the value in the *EPC* (*ErrorEPC* for errors, or *DEPC* for debug exceptions) is sufficient to restart execution. It also ensures that exceptions are taken in the order of execution; an instruction taking an exception may itself be killed by an instruction further down the pipeline that takes an exception in a later cycle.

## 4.2 Exception Priority

Table 4.1 contains a list and a brief description of all exception conditions. The exceptions are listed in the order of their relative priority, from highest priority (Reset) to lowest priority. When several exceptions occur simultaneously, the exception with the highest priority is taken.

**Table 4.1 Priority of Exceptions**

Exception	Description
Reset	Assertion of <i>SI_ColdReset</i> signal.
Soft Reset	Assertion of <i>SI_Reset</i> signal.
DSS	EJTAG Debug Single Step.
DINT	EJTAG Debug Interrupt. Caused by the assertion of the external <i>EJ_DINT</i> input, or by setting the <i>EjtagBrk</i> bit in the <i>ECR</i> register.
NMI	Asserting edge of <i>SI_NMI</i> signal.
Interrupt	Assertion of unmasked hardware or software interrupt signal.
Protection - Instruction fetch	Instruction fetch access to a protected memory region was attempted.
DIB	EJTAG debug hardware instruction break matched.
AdEL	Fetch address alignment error. User mode fetch reference to kernel address.
ISRAM Parity Error	Parity error on I-SRAM access
IBE	Instruction fetch bus error.
Instruction Validity Exceptions	An instruction could not be completed because it was not allowed access to the required resources (Coprocessor Unusable) or was illegal (Reserved Instruction). If both exceptions occur on the same instruction, the Coprocessor Unusable Exception takes priority over the Reserved Instruction Exception.
Protection - Instr Execution	Attempted to write EBase when not allowed by MPU..
Execution Exception	An instruction-based exception occurred: Integer overflow, trap, system call, breakpoint, floating point, or Coprocessor 2 exception.
Tr	Execution of a trap (when trap condition is true).
Protection - Data access	Data access to a protected memory region was attempted.
DDBL / DDBS	EJTAG Data Address Break (address only) or EJTAG Data Value Break on Store (address and value).
AdEL	Load address alignment error. User mode load reference to kernel address.
AdES	Store address alignment error. User mode store to kernel address.



Table 4.1 Priority of Exceptions (Continued)

Exception	Description
DSRAM Parity Error	Parity error on D-SRAM access
DBE	Load or store bus error.
DDBL	EJTAG data hardware breakpoint matched in load data compare.
CBrk	EJTAG complex breakpoint.

## 4.3 Interrupts

In the MIPS32® Release 1 architecture, support for exceptions included two software interrupts, six hardware interrupts, and a special-purpose timer interrupt. The timer interrupt was provided external to the *CoreType-Ic* and was typically combined with hardware interrupt 5 in a system-dependent manner. Interrupts were handled either through the general exception vector (offset 0x180) or the special interrupt vector (0x200), based on the value of *CauseIV*. Software was required to prioritize interrupts as a function of the *CauseIV* bits in the interrupt handler prologue.

Release 2 of the Architecture, implemented by the M14K core, adds a number of upward-compatible extensions to the Release 1 interrupt architecture, including support for vectored interrupts and the implementation of a new interrupt mode that permits the use of an external interrupt controller.

The M14K core also includes the Microcontroller Application-Specific Extension (MCUASE) that provides enhanced interrupt delivery and interrupt-latency reduction.

### 4.3.1 Interrupt Modes

The M14K core includes support for three interrupt modes, as defined by Release 2 of the Architecture:

- Interrupt Compatibility mode, in which the behavior of the M14K is identical to the behavior of a Release 1 implementations.
- Vectored Interrupt (VI) mode, which adds the ability to prioritize and vector interrupts to a handler dedicated to that interrupt, and to assign a GPR shadow set for use during interrupt processing. The presence of this mode is denoted by the *VInt* bit in the *Config3* register. Although this mode is architecturally optional, it is always present on the M14K *CoreType-Ic*, so the *VInt* bit will always read as a 1.
- External Interrupt Controller (EIC) mode, which redefines the way interrupts are handled to provide full support for an external interrupt controller that handles prioritization and vectoring of interrupts. As with VI mode, this mode is architecturally optional. The presence of this mode is denoted by the *VEIC* bit in the *Config3* register. On the M14K *CoreType-Ic*, the *VEIC* bit is set externally by the static input, *SI\_EICPresent*, to allow system logic to indicate the presence of an external interrupt controller.

Following reset, the M14K processor defaults to Compatibility mode, which is fully compatible with all implementations of Release 1 of the Architecture.

Table 4.2 shows the current interrupt mode of the processor as a function of the Coprocessor 0 register fields that can affect the mode.

**Table 4.2 Interrupt Modes**

Status <sub>BEV</sub>	Cause <sub>IV</sub>	IntCtl <sub>VS</sub>	Config3 <sub>VINT</sub>	Config3 <sub>VEIC</sub>	Interrupt Mode
1	x	x	x	x	Compatibility
x	0	x	x	x	Compatibility
x	x	=0	x	x	Compatibility
0	1	≠0	1	0	Vectored Interrupt
0	1	≠0	x	1	External Interrupt Controller
0	1	≠0	0	0	Can't happen - IntCtl <sub>VS</sub> can not be non-zero if neither Vectored Interrupt nor External Interrupt Controller mode is implemented.
"x" denotes don't care					

#### 4.3.1.1 Interrupt Compatibility Mode

This is the default interrupt mode for the processor and is entered when a Reset exception occurs. In this mode, interrupts are non-vectored and dispatched through exception vector offset 16#180 (if Cause<sub>IV</sub> = 0) or vector offset 16#200 (if Cause<sub>IV</sub> = 1). This mode is in effect if any of the following conditions are true:

- Cause<sub>IV</sub> = 0
- Status<sub>BEV</sub> = 1
- IntCtl<sub>VS</sub> = 0, which would be the case if vectored interrupts are not implemented, or have been disabled.

Here is a typical software handler for interrupt compatibility mode:

```

/*
 * Assumptions:
 * - CauseIV = 1 (if it were zero, the interrupt exception would have to
 *   be isolated from the general exception vector before getting
 *   here)
 * - GPRs k0 and k1 are available (no shadow register switches invoked in
 *   compatibility mode)
 * - The software priority is IP9..IP0 (HW7..HW0, SW1..SW0)
 *
 * Location: Offset 0x200 from exception base
 */

IVexception:
    mfc0    k0, C0_Cause      /* Read Cause register for IP bits */
    mfc0    k1, C0_Status     /* and Status register for IM bits */
    andi    k0, k0, M_CauseIM /* Keep only IP bits from Cause */
    and     k0, k0, k1        /* and mask with IM bits */
    beq     k0, zero, Dismiss /* no bits set - spurious interrupt */
    clz     k0, k0            /* Find first bit set, IP9..IP0; k0 = 14..23 */

```

```

xori    k0, k0, 0x17          /* 14..23 => 9..0 */
sll     k0, k0, VS            /* Shift to emulate software IntCtlVS */
la      k1, VectorBase        /* Get base of 10 interrupt vectors */
addu    k0, k0, k1            /* Compute target from base and offset */
jr      k0                    /* Jump to specific exception routine */
nop

/*
 * Each interrupt processing routine processes a specific interrupt, analogous
 * to those reached in VI or EIC interrupt mode. Since each processing routine
 * is dedicated to a particular interrupt line, it has the context to know
 * which line was asserted. Each processing routine may need to look further
 * to determine the actual source of the interrupt if multiple interrupt requests
 * are ORed together on a single IP line. Once that task is performed, the
 * interrupt may be processed in one of two ways:
 *
 * - Completely at interrupt level (e.g., a simply UART interrupt). The
 *   SimpleInterrupt routine below is an example of this type.
 * - By saving sufficient state and re-enabling other interrupts. In this
 *   case the software model determines which interrupts are disabled during
 *   the processing of this interrupt. Typically, this is either the single
 *   StatusIM bit that corresponds to the interrupt being processed, or some
 *   collection of other StatusIM bits so that "lower" priority interrupts are
 *   also disabled. The NestedInterrupt routine below is an example of this type.
 */

SimpleInterrupt:
/*
 * Process the device interrupt here and clear the interrupt request
 * at the device. In order to do this, some registers may need to be
 * saved and restored. The coprocessor 0 state is such that an ERET
 * will simply return to the interrupted code.
 */
    eret                      /* Return to interrupted code */

NestedException:
/*
 * Nested exceptions typically require saving the EPC and Status registers,
 * any GPRs that may be modified by the nested exception routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

    /* Save GPRs here, and setup software context */
    mfc0   k0, C0_EPC          /* Get restart address */
    sw     k0, EPCTSave        /* Save in memory */
    mfc0   k0, C0_Status        /* Get Status value */
    sw     k0, StatusSave       /* Save in memory */
    li     k1, ~IMbitsToClear   /* Get Im bits to clear for this interrupt */
    /*      this must include at least the IM bit */
    /*      for the current interrupt, and may include */
    /*      others */
    and     k0, k0, k1          /* Clear bits in copy of Status */
    ins     k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
    /* Clear KSU, ERL, EXL bits in k0 */
    mtc0    k0, C0_Status       /* Modify mask, switch to kernel mode, */

```

```

/* re-enable interrupts */

/*
 * Process interrupt here, including clearing device interrupt.
 * In some environments this may be done with a thread running in
 * kernel or user mode. Such an environment is well beyond the scope of
 * this example.
 */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

di          /* Disable interrupts - may not be required */
lw    k0, StatusSave /* Get saved Status (including EXL set) */
lw    k1, EPCSave    /* and EPC */
mtc0   k0, C0_Status /* Restore the original value */
mtc0   k1, C0_EPC    /* and EPC */
/* Restore GPRs and software state */
eret      /* Dismiss the interrupt */

```

### 4.3.1.2 Vectored Interrupt (VI) Mode

In Vectored Interrupt (VI) mode, a priority encoder prioritizes pending interrupts and generates a vector which can be used to direct each interrupt to a dedicated handler routine. This mode also allows each interrupt to be mapped to a GPR shadow register set for use by the interrupt handler. VI mode is in effect when all the following conditions are true:

- $Config3_{VInt} = 1$
- $Config3_{VEIC} = 0$
- $IntCtl_{VS} \neq 0$
- $Cause_{IV} = 1$
- $Status_{BEV} = 0$

In VI interrupt mode, the eight hardware interrupts are interpreted as individual hardware interrupt requests. The timer interrupt is combined in a system-dependent way (external to the core) with the hardware interrupts (the interrupt with which they are combined is indicated by the *PTI* field in *IntCtl*) to provide the appropriate relative priority of the timer interrupt with that of the hardware interrupts. The processor interrupt logic ANDs each of the  $Cause_{IP}$  bits with the corresponding  $Status_{IM}$  bits. If any of these values is 1, and if interrupts are enabled ( $Status_{IE} = 1$ ,

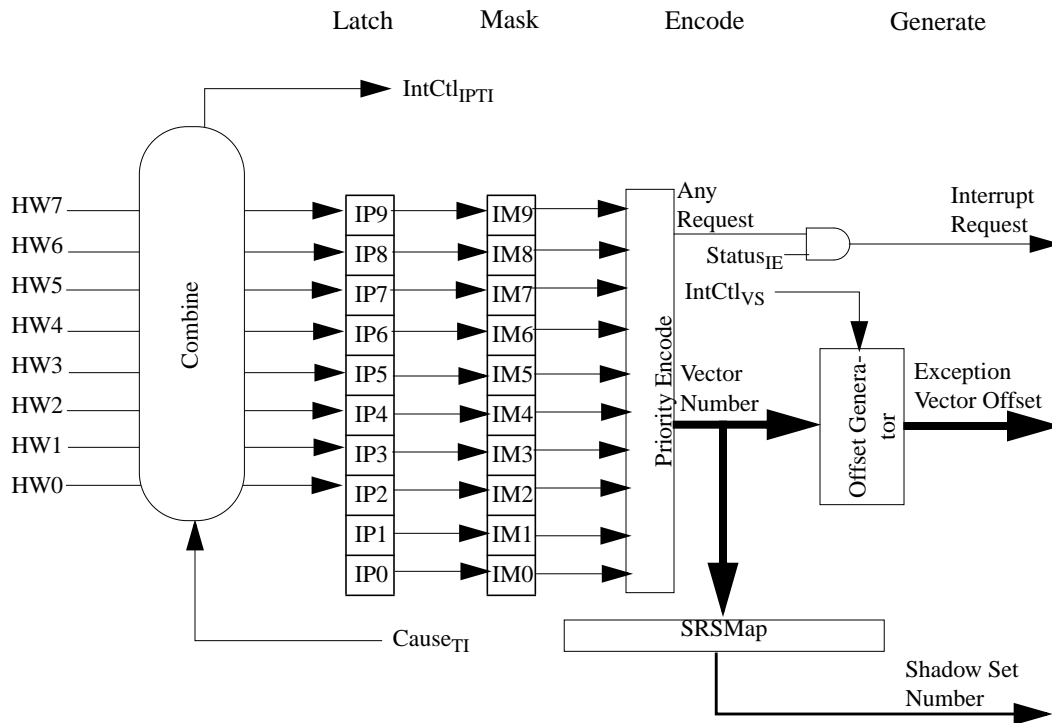
$Status_{EXL} = 0$ , and  $Status_{ERL} = 0$ ), an interrupt is signaled and a priority encoder scans the values in the order shown in Table 4.3.

**Table 4.3 Relative Interrupt Priority for Vectored Interrupt Mode**

Relative Priority	Interrupt Type	Interrupt Source	Interrupt Request Calculated From	Vector Number Generated by Priority Encoder
Highest Priority	Hardware	HW7	IP9 and IM9	9
		HW6	IP8 and IM8	8
		HW5	IP7 and IM7	7
		HW4	IP6 and IM6	6
		HW3	IP5 and IM5	5
		HW2	IP4 and IM4	4
		HW1	IP3 and IM3	3
		HW0	IP2 and IM2	2
Lowest Priority	Software	SW1	IP1 and IM1	1
		SW0	IP0 and IM0	0

The priority order places a relative priority on each hardware interrupt and places the software interrupts at a priority lower than all hardware interrupts. When the priority encoder finds the highest priority pending interrupt, it outputs an encoded vector number that is used in the calculation of the handler for that interrupt, as described below. This is shown pictorially in Figure 4.1.

Figure 4.1 Interrupt Generation for Vectored Interrupt Mode



A typical software handler for vectored interrupt mode bypasses the entire sequence of code following the IV exception label shown for the compatibility mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine. Unlike the compatibility mode examples, a vectored interrupt handler may take advantage of a dedicated GPR shadow set to avoid saving any registers. As such, the Simple Interrupt code shown above need not save the GPRs.

A nested interrupt is similar to that shown for compatibility mode, but may also take advantage of running the nested exception routine in the GPR shadow set dedicated to the interrupt or in another shadow set. Such a routine might look as follows:

```
NestedException:
/*
 * Nested exceptions typically require saving the EPC, Status and SRSSctl registers,
 * setting up the appropriate GPR shadow set for the routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

/* Use the current GPR shadow set, and setup software context */
mfc0 k0, C0_EPC          /* Get restart address */
sw    k0, EPCSave        /* Save in memory */
mfc0 k0, C0_Status       /* Get Status value */
sw    k0, StatusSave     /* Save in memory */
mfc0 k0, C0_SRSSctl      /* Save SRSSctl if changing shadow sets */
sw    k0, SRSSctlSave
```

```

li      k1, ~IMbitsToClear /* Get Im bits to clear for this interrupt */
                                /* this must include at least the IM bit */
                                /* for the current interrupt, and may include */
                                /* others */
and     k0, k0, k1          /* Clear bits in copy of Status */
/* If switching shadow sets, write new value to SRSCtlpss here */
ins     k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                /* Clear KSU, ERL, EXL bits in k0 */
mtc0    k0, C0_Status        /* Modify mask, switch to kernel mode, */
                                /* re-enable interrupts */

/*
 * If switching shadow sets, clear only KSU above, write target
 * address to EPC, and do execute an eret to clear EXL, switch
 * shadow sets, and jump to routine
 */

/* Process interrupt here, including clearing device interrupt */

/*
 * To complete interrupt processing, the saved values must be restored
 * and the original interrupted code restarted.
 */

di      /* Disable interrupts - may not be required */
lw      k0, StatusSave       /* Get saved Status (including EXL set) */
lw      k1, EPCSave          /* and EPC */
mtc0    k0, C0_Status        /* Restore the original value */
lw      k0, SRSCtlSave       /* Get saved SRSCtl */
mtc0    k1, C0_EPC           /* and EPC */
mtc0    k0, C0_SRSCtl        /* Restore shadow sets */
ehb     /* Clear hazard */
eret    /* Dismiss the interrupt */

```

#### 4.3.1.3 External Interrupt Controller Mode

External Internal Interrupt Controller Mode redefines the way that the processor interrupt logic is configured to provide support for an external interrupt controller. The interrupt controller is responsible for prioritizing all interrupts, including hardware, software, timer, and performance counter interrupts, and directly supplying to the processor the priority level and vector number of the highest priority interrupt. EIC interrupt mode is in effect if all of the following conditions are true:

- $Config3_{VEIC} = 1$
- $IntCtl_{VS} \neq 0$
- $Cause_{IV} = 1$
- $Status_{BEV} = 0$

In EIC interrupt mode, the processor sends the state of the software interrupt requests ( $Cause_{IP1..IP0}$ ), the timer interrupt request ( $Cause_{TI}$ ), the performance counter interrupt request ( $Cause_{PCI}$ ) and Fast Debug Channel Interrupt ( $Cause_{FDCI}$ ) to the external interrupt controller, where it prioritizes these interrupts in a system-dependent way with other hardware interrupts. The interrupt controller can be a hard-wired logic block, or it can be configurable based on control and status registers. This allows the interrupt controller to be more specific or more general as a function of the system environment and needs.

The external interrupt controller prioritizes its interrupt requests and produces the priority level and the vector number of the highest priority interrupt to be serviced. The priority level, called the Requested Interrupt Priority Level (RIPL), is an 8-bit encoded value in the range 0..255, inclusive. A value of 0 indicates that no interrupt requests are pending. The values 1..255 represent the lowest (1) to highest (255) RIPL for the interrupt to be serviced. The interrupt controller passes this value on the 8 hardware interrupt lines, which are treated as an encoded value in EIC interrupt mode. There are two implementation options available for the vector offset:

1. The first option is to send a separate vector number along with the RIPL to the processor.
2. A second option is to send an entire vector offset along with the RIPL to the processor. This option is enabled through the core's configuration GUI, and it is not affected by software.

The M14K core does not support the option to treat the RIPL value as the vector number for the processor.

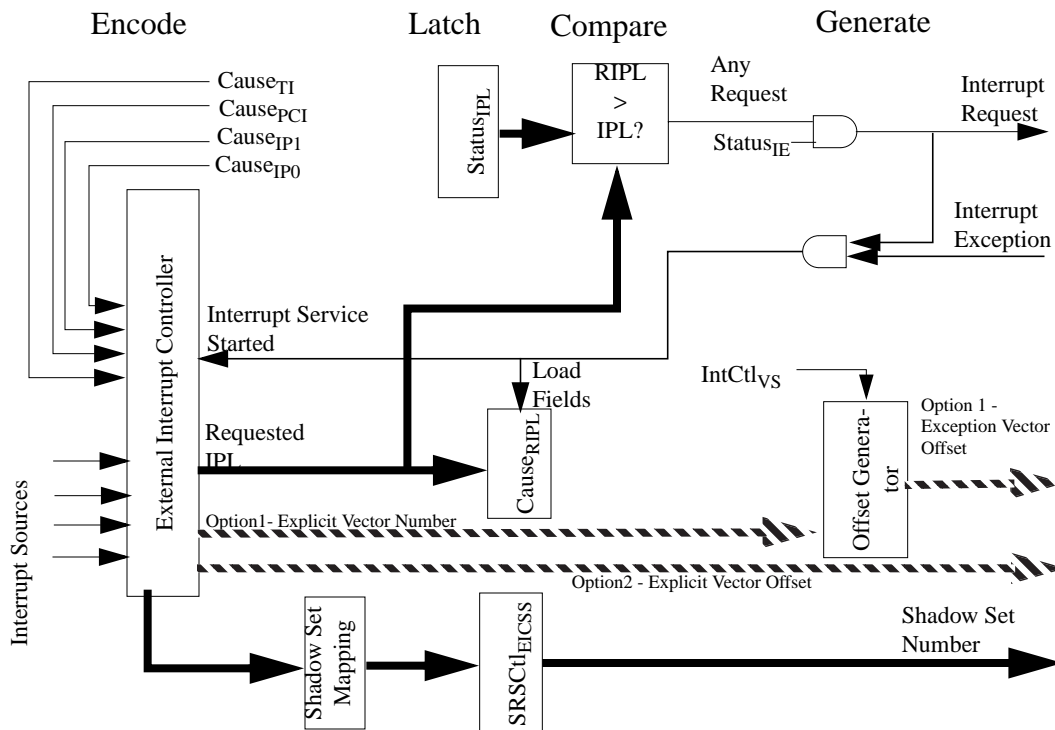
*Status*<sub>IPL</sub> (which overlays *Status*<sub>M9..IM2</sub>) is interpreted as the Interrupt Priority Level (IPL) at which the processor is currently operating (with a value of zero indicating that no interrupt is currently being serviced). When the interrupt controller requests service for an interrupt, the processor compares *RIPL* with *Status*<sub>IPL</sub> to determine if the requested interrupt has higher priority than the current *IPL*. If RIPL is strictly greater than *Status*<sub>IPL</sub>, and interrupts are enabled (*Status*<sub>IE</sub> = 1, *Status*<sub>EXL</sub> = 0, and *Status*<sub>ERL</sub> = 0) an interrupt request is signaled to the pipeline. When the processor starts the interrupt exception, it loads RIPL into *Cause*<sub>RIPL</sub> (which overlays *Cause*<sub>IP9..IP2</sub>) and signals the external interrupt controller to notify it that the request is being serviced. Because *Cause*<sub>RIPL</sub> is only loaded by the processor when an interrupt exception is signaled, it is available to software during interrupt processing. The vector number that the EIC passes to the core is combined with the *IntCtl*<sub>VS</sub> to determine where the interrupt service routine is located. The vector number is not stored in any software-visible registers.

In EIC interrupt mode, the external interrupt controller is also responsible for supplying the GPR shadow set number to use when servicing the interrupt. As such, the *SRSMap* register is not used in this mode, and the mapping of the vectored interrupt to a GPR shadow set is done by programming (or designing) the interrupt controller to provide the correct GPR shadow set number when an interrupt is requested. When the processor loads an interrupt request into *Cause*<sub>RIPL</sub>, it also loads the GPR shadow set number into *SRSCtl*<sub>EICSS</sub>, which is copied to *SRSCtl*<sub>CSS</sub> when the interrupt is serviced.

The operation of EIC interrupt mode is shown pictorially in [Figure 4.2](#).



Figure 4.2 Interrupt Generation for External Interrupt Controller Interrupt Mode



A typical software handler for EIC interrupt mode bypasses the entire sequence of code following the IV exception label shown for the compatibility-mode handler above. Instead, the hardware performs the prioritization, dispatching directly to the interrupt processing routine. Unlike the compatibility mode examples, an EIC interrupt handler may take advantage of a dedicated GPR shadow set to avoid saving any registers. As such, the Simple Interrupt code shown above need not save the GPRs.

A nested interrupt is similar to that shown for compatibility mode, but may also take advantage of running the nested exception routine in the GPR shadow set dedicated to the interrupt or in another shadow set. It also need only copy  $Cause_{RIPL}$  to  $Status_{IPL}$  to prevent lower priority interrupts from interrupting the handler. Such a routine might look as follows:

```
NestedException:
/*
 * Nested exceptions typically require saving the EPC, Status, and SRSCtl registers,
 * setting up the appropriate GPR shadow set for the routine, disabling
 * the appropriate IM bits in Status to prevent an interrupt loop, putting
 * the processor in kernel mode, and re-enabling interrupts. The sample code
 * below can not cover all nuances of this processing and is intended only
 * to demonstrate the concepts.
 */

/* Use the current GPR shadow set, and setup software context */
mfc0 k1, C0_Cause      /* Read Cause to get RIPL value */
mfc0 k0, C0_EPC        /* Get restart address */
srl  k1, k1, S_CauseRIPL /* Right justify RIPL field */
sw   k0, EPCSave       /* Save in memory */
mfc0 k0, C0_Status     /* Get Status value */
```

```

sw      k0, StatusSave      /* Save in memory */
ins     k0, k1, S_StatusIPL, 6 /* Set IPL to RIPL in copy of Status */
mfc0    k1, C0_SRSCtl       /* Save SRSCtl if changing shadow sets */
sw      k1, SRSCtlSave
/* If switching shadow sets, write new value to SRSCtlpsg here */
ins     k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
/* Clear KSU, ERL, EXL bits in k0 */
mtc0    k0, C0_Status       /* Modify IPL, switch to kernel mode, */
/* re-enable interrupts */

/*
 * If switching shadow sets, clear only KSU above, write target
 * address to EPC, and do execute an eret to clear EXL, switch
 * shadow sets, and jump to routine
 */

/* Process interrupt here, including clearing device interrupt */

/*
 * The interrupt completion code is identical to that shown for VI mode above.
 */

```

### 4.3.2 Generation of Exception Vector Offsets for Vectored Interrupts

For vectored interrupts (in either VI or EIC interrupt mode), a vector number is produced by the interrupt control logic. This number is combined with *IntCtl<sub>VS</sub>* to create the interrupt offset, which is added to 16#200 to create the exception vector offset. For VI interrupt mode, the vector number is in the range 0..9, inclusive. For EIC interrupt mode, the vector number is in the range 0..63, inclusive. The *IntCtl<sub>VS</sub>* field specifies the spacing between vector locations. If this value is zero (the default reset state), the vector spacing is zero and the processor reverts to Interrupt Compatibility Mode. A non-zero value enables vectored interrupts, and Table 4.4 shows the exception vector offset for a representative subset of the vector numbers and values of the *IntCtl<sub>VS</sub>* field.

**Table 4.4 Exception Vector Offsets for Vectored Interrupts**

Vector Number	Value of IntCtl <sub>VS</sub> Field				
	2#00001	2#00010	2#00100	2#01000	2#10000
0	16#0200	16#0200	16#0200	16#0200	16#0200
1	16#0220	16#0240	16#0280	16#0300	16#0400
2	16#0240	16#0280	16#0300	16#0400	16#0600
3	16#0260	16#02C0	16#0380	16#0500	16#0800
4	16#0280	16#0300	16#0400	16#0600	16#0A00
5	16#02A0	16#0340	16#0480	16#0700	16#0C00
6	16#02C0	16#0380	16#0500	16#0800	16#0E00
7	16#02E0	16#03C0	16#0580	16#0900	16#1000
		•			
		•			
		•			
61	16#09A0	16#1140	16#2080	16#3F00	16#7C00
62	16#09C0	16#1180	16#2100	16#4000	16#7E00
63	16#09E0	16#11C0	16#2180	16#4100	16#8000

The general equation for the exception vector offset for a vectored interrupt is:

$$\text{vectorOffset} \leftarrow 16\#200 + (\text{vectorNumber} \times (\text{IntCtlVs} \parallel 2\#00000))$$

When using large vector spacing and EIC mode, the offset value can overlap with bits that are specified in the EBase register. Software must ensure that any overlapping bits are specified as 0 in EBase. This implementation ORs together the offset and base registers, but it is architecturally undefined and software should not rely on this behavior.

Although there are 255 EIC priority interrupts, only 64 vectors are provided. There is no one-to-one mapping for each EIC interrupt to its interrupt vector. The 255 priority interrupts will share the 64 interrupt vectors as specified by the *SL\_EICVector*[5:0] input pins. However, as mentioned in option 2 of [Section 4.3.1.3 “External Interrupt Controller Mode”](#), the *SL\_Offse*[17:1] input pins can be used to provide each EIC interrupt with a unique interrupt handler location.

### 4.3.3 MCU ASE Enhancement for Interrupt Handling

The MCU ASE extends the microMIPS/MIPS32 Architecture with a set of new features designed for the microcontroller market. The MCU ASE contains enhancements in two key areas: interrupt delivery and interrupt latency. For more details, refer to the *The MCU Privileged Resource Architecture*” chapter of the *MIPS32 Architecture for Programmers Volume IV-h: The MuCon Application-Specific Extension to the MIPS32 and microMIPS32 Architectures* [10].

#### 4.3.3.1 Interrupt Delivery

The MCU ASE extends the number of hardware interrupt sources from 6 to 8. For legacy and vectored-interrupt mode, this represents 8 external interrupt sources. For EIC mode, the widened *IPL* and *RIPL* fields can now represent 256 external interrupt sources.

#### 4.3.3.2 Interrupt Latency Reduction

The MCU ASE includes a package of extensions to microMIPS/MIPS32 that decrease the latency of the processor’s response to a signalled interrupt.

##### ***Interrupt Vector Prefetching***

Normally on MIPS architecture processors, when an interrupt or exception is signalled, execution pipelines must be flushed before the interrupt/exception handler is fetched. This is necessary to avoid mixing the contexts of the interrupted/faulting program and the exception handler. The MCU ASE introduces a hardware mechanism in which the interrupt exception vector is prefetched whenever the interrupt input signals change. The prefetch memory transaction occurs in parallel with the pipeline flush and exception prioritization. This decreases the overall latency of the execution of the interrupt handler’s first instruction.

##### ***Automated Interrupt Prologue***

The use of Shadow Register Sets avoids the software steps of having to save general-purpose registers before handling an interrupt.

The MCU ASE adds additional hardware logic that automatically saves some of the COP0 state in the stack and automatically updates some of the COP0 registers in preparation for interrupt handling.

### ***Automated Interrupt Epilogue***

A mirror to the Automated Prologue, this feature automates the restoration of some of the COP0 registers from the stack and the preparation of some of the COP0 registers for returning to non-exception mode. This feature is implemented within the IRET instruction, which is introduced in this ASE.

### ***Interrupt Chaining***

An optional feature of the Automated Interrupt Epilogue, this feature allows handling a second interrupt after a primary interrupt is handled, without returning to non-exception mode (and the related pipeline flushes that would normally be necessary).

## **4.4 GPR Shadow Registers**

Release 2 of the Architecture optionally removes the need to save and restore GPRs on entry to high priority interrupts or exceptions, and to provide specified processor modes with the same capability. This is done by introducing multiple copies of the GPRs, called *shadow sets*, and allowing privileged software to associate a shadow set with entry to kernel mode via an interrupt vector or exception. The normal GPRs are logically considered shadow set zero.

The number of GPR shadow sets is a build-time option on the M14K core. Although Release 2 of the Architecture defines a maximum of 16 shadow sets, the core allows one (the normal GPRs), two, four, eight or sixteen shadow sets. The highest number actually implemented is indicated by the  $SRSCtl_{HSS}$  field. If this field is zero, only the normal GPRs are implemented.

Shadow sets are new copies of the GPRs that can be substituted for the normal GPRs on entry to kernel mode via an interrupt or exception. Once a shadow set is bound to a kernel mode entry condition, reference to GPRs work exactly as one would expect, but they are redirected to registers that are dedicated to that condition. Privileged software may need to reference all GPRs in the register file, even specific shadow registers that are not visible in the current mode. The RDPGPR and WRPGPR instructions are used for this purpose. The  $CSS$  field of the  $SRSCtl$  register provides the number of the current shadow register set, and the  $PSS$  field of the  $SRSCtl$  register provides the number of the previous shadow register set (that which was current before the last exception or interrupt occurred).

If the processor is operating in VI interrupt mode, binding of a vectored interrupt to a shadow set is done by writing to the  $SRSMap$  register. If the processor is operating in EIC interrupt mode, the binding of the interrupt to a specific shadow set is provided by the external interrupt controller, and is configured in an implementation-dependent way. Binding of an exception or non-vectored interrupt to a shadow set is done by writing to the  $ESS$  field of the  $SRSCtl$  register. When an exception or interrupt occurs, the value of  $SRSCtl_{CSS}$  is copied to  $SRSCtl_{PSS}$ , and  $SRSCtl_{CSS}$  is set to the value taken from the appropriate source. On an ERET, the value of  $SRSCtl_{PSS}$  is copied back into  $SRSCtl_{CSS}$  to restore the shadow set of the mode to which control returns. More precisely, the rules for updating the fields in the  $SRSCtl$  register on an interrupt or exception are as follows:

1. No field in the  $SRSCtl$  register is updated if any of the following conditions is true. In this case, steps 2 and 3 are skipped.
  - The exception is one that sets  $Status_{ERL}$ : Reset, Soft Reset, or NMI.
  - The exception causes entry into EJTAG Debug Mode.
  - $Status_{BEV} = 1$
  - $Status_{EXL} = 1$

2.  $SRSCtl_{CSS}$  is copied to  $SRSCtl_{PSS}$ .
3.  $SRSCtl_{CSS}$  is updated from one of the following sources:
  - The appropriate field of the  $SRSSMap$  register, based on IPL, if the exception is an interrupt,  $Cause_{IV} = 1$ ,  $Config3_{VEIC} = 0$ , and  $Config3_{VInt} = 1$ . These are the conditions for a vectored interrupt.
  - The  $EICSS$  field of the  $SRSCtl$  register if the exception is an interrupt,  $Cause_{IV} = 1$ , and  $Config3_{VEIC} = 1$ . These are the conditions for a vectored EIC interrupt.
  - The  $ESS$  field of the  $SRSCtl$  register in any other case. This is the condition for a non-interrupt exception, or a non-vectored interrupt.

Similarly, the rules for updating the fields in the  $SRSCtl$  register at the end of an exception or interrupt are as follows:

1. No field in the  $SRSCtl$  register is updated if any of the following conditions is true. In this case, step 2 is skipped.
  - A DERET is executed.
  - An ERET is executed with  $Status_{ERL} = 1$ .
2.  $SRSCtl_{PSS}$  is copied to  $SRSCtl_{CSS}$ .

These rules have the effect of preserving the  $SRSCtl$  register in any case of a nested exception or one which occurs before the processor has been fully initialize ( $Status_{BEV} = 1$ ).

Privileged software may switch the current shadow set by writing a new value into  $SRSCtl_{PSS}$ , loading  $EPC$  with a target address, and doing an ERET.

## 4.5 Exception Vector Locations

The Reset, Soft Reset, and NMI exceptions are always vectored to location 16#BFC0.0000. EJTAG Debug exceptions are vectored to location 16#BFC0.0480, or to location 16#FF20.0200 if the ProbTrap bit is zero or one, respectively, in the *EJTAG\_Control\_register*. Addresses for all other exceptions are a combination of a vector offset and a vector base address. In Release 1 of the architecture, the vector base address was fixed. In Release 2 of the architecture, software is allowed to specify the vector base address via the *EBase* register for exceptions that occur when  $Status_{BEV}$  equals 0. Table 4.5 gives the vector base address as a function of the exception and whether the *BEV* bit is set in the *Status* register. Table 4.6 gives the offsets from the vector base address as a function of the exception. Note that the *IV* bit in the *Cause* register causes Interrupts to use a dedicated exception vector offset, rather than the general exception vector. For implementations of Release 2 of the Architecture,

Table 4.4 shows the offset from the base address in the case where  $Status_{BEV} = 0$  and  $Cause_{IV} = 1$ . For implementations of Release 1 of the architecture in which  $Cause_{IV} = 1$ , the vector offset is as if  $IntCtl_{VS}$  were 0. Table 4.7 combines these two tables into one that contains all possible vector addresses as a function of the state that can affect the

vector selection. To avoid complexity in the table, the vector address value assumes that the *EBase* register, as implemented in Release 2 devices, is not changed from its reset state and that *IntCtl<sub>VS</sub>* is 0.

**Table 4.5 Exception Vector Base Addresses**

Exception	Status <sub>BEV</sub>	
	0	1
Reset, Soft Reset, NMI	16#BFC0.0000	
EJTAG Debug (with <i>ProbEn</i> = 0 in the <i>EJTAG Control Register</i> )	16#BFC0.0480	
EJTAG Debug (with <i>ProbEn</i> = 1 in the <i>EJTAG Control Register</i> )	16#FF20.0200	
SRAM Parity Error	EBase <sub>31..30</sub>    1    EBase <sub>28..12</sub>    16#000 Note that EBase <sub>31..30</sub> have the fixed value 2#10	16#BFC0.0300
Other	<i>For Release 1 of the architecture:</i> 16#8000.0000 <i>For Release 2 of the architecture:</i> EBase <sub>31..12</sub>    16#000 Note that EBase <sub>31..30</sub> have the fixed value 2#10	16#BFC0.0200

**Table 4.6 Exception Vector Offsets**

Exception	Vector Offset
General Exception	16#180
Interrupt, <i>Cause<sub>IV</sub></i> = 1	16#200 (In Release 2 implementations, this is the base of the vectored interrupt table when <i>Status<sub>BEV</sub></i> = 0)
Reset, Soft Reset, NMI	None (Uses Reset Base Address)

**Table 4.7 Exception Vectors**

Exception	Status <sub>BEV</sub>	Status <sub>EXL</sub>	Cause <sub>IV</sub>	EJTAG ProbEn	Vector  For Release 2 Implementations, assumes that EBase retains its reset state and that IntCtl <sub>VS</sub> = 0
Reset, Soft Reset, NMI	x	x	x	x	16#BFC0.0000
EJTAG Debug	x	x	x	0	16#BFC0.0480
EJTAG Debug	x	x	x	1	16#FF20.0200
SRAM Parity Error	0	x	x	x	16#EBase[31:30]    2#1    EBase[28:12]    16#100
SRAM Parity Error	1	x	x	x	16#BFC0.0300
Interrupt	0	0	0	x	16#8000.0180

Table 4.7 Exception Vectors (Continued)

Exception	Status <sub>BEV</sub>	Status <sub>EXL</sub>	Cause <sub>IV</sub>	EJTAG ProbEn	Vector  For Release 2 Implementations, assumes that EBase retains its reset state and that IntCtl <sub>VS</sub> = 0
Interrupt	0	0	1	x	16#8000.0200
Interrupt	1	0	0	x	16#BFC0.0380
Interrupt	1	0	1	x	16#BFC0.0400
All others	0	x	x	x	16#8000.0180
All others	1	x	x	x	16#BFC0.0380
'x' denotes don't care					

## 4.6 General Exception Processing

With the exception of Reset, Soft Reset, NMI, cache error, and EJTAG Debug exceptions, which have their own special processing as described below, exceptions have the same basic processing flow:

- If the *EXL* bit in the *Status* register is zero, the *EPC* register is loaded with the PC at which execution will be restarted and the *BD* bit is set appropriately in the *Cause* register (see Table 5.15). The value loaded into the *EPC* register is dependent on whether the processor implements microMIPS, and whether the instruction is in the delay slot of a branch or jump which has delay slots. Table 4.8 shows the value stored in each of the CP0 PC registers, including *EPC*. For implementations of Release 2 of the Architecture if *Status<sub>BEV</sub>* = 0, the *CSS* field in the *SRSCtl* register is copied to the *PSS* field, and the *CSS* value is loaded from the appropriate source.

If the *EXL* bit in the *Status* register is set, the *EPC* register is not loaded and the *BD* bit is not changed in the *Cause* register. For implementations of Release 2 of the Architecture, the *SRSCtl* register is not changed.

Table 4.8 Value Stored in EPC, ErrorEPC, or DEPC on an Exception

microMIPS Implemented?	In Branch/Jump Delay Slot?	Value stored in EPC/ErrorEPC/DEPC
No	No	Address of the instruction
No	Yes	Address of the branch or jump instruction (PC-4)
Yes	No	Upper 31 bits of the address of the instruction, combined with the <i>ISA Mode</i> bit
Yes	Yes	Upper 31 bits of the branch or jump instruction (PC-2 or PC-4 depending on size of the instruction in the microMIPS <i>ISA Mode</i> and PC-4 in the 32-bit <i>ISA Mode</i> ), combined with the <i>ISA Mode</i> bit

- The *CE* and *ExcCode* fields of the *Cause* registers are loaded with the values appropriate to the exception. The *CE* field is loaded, but not defined, for any exception type other than a coprocessor unusable exception.
- The *EXL* bit is set in the *Status* register.
- The processor is started at the exception vector.

The value loaded into *EPC* represents the restart address for the exception and need not be modified by exception handler software in the normal case. Software need not look at the *BD* bit in the *Cause* register unless it wishes to identify the address of the instruction that actually caused the exception.

Note that individual exception types may load additional information into other registers. This is noted in the description of each exception type below.

### Operation:

```

/* If StatusEXL is 1, all exceptions go through the general exception vector */
/* and neither EPC nor CauseBD nor SRSCtl are modified */
if StatusEXL = 1 then
    vectorOffset ← 16#180
else
    if InstructionInBranchDelaySlot then
        EPC ← restartPC /* PC of branch/jump */
        CauseBD ← 1
    else
        EPC ← restartPC /* PC of instruction */
        CauseBD ← 0
    endif
endif

/* Compute vector offsets as a function of the type of exception */
NewShadowSet ← SRSCtlESS /* Assume exception, Release 2 only */
if ExceptionType = TLBRefill then
    vectorOffset ← 16#000
elseif (ExceptionType = Interrupt) then
    if (CauseIV = 0) then
        vectorOffset ← 16#180
    else
        if (StatusBEV = 1) or (IntCtlVS = 0) then
            vectorOffset ← 16#200
        else
            if Config3VEIC = 1 then
                VecNum ← CauseRIPL
                NewShadowSet ← SRSCtlEICSS
            else
                VecNum ← VIntPriorityEncoder()
                NewShadowSet ← SRSCtlIPL × 4 + 3..IPL × 4
            endif
            vectorOffset ← 16#200 + (VecNum × (IntCtlVS || 2#00000))
        endif /* if (StatusBEV = 1) or (IntCtlVS = 0) then */
    endif /* if (CauseIV = 0) then */
endif /* elseif (ExceptionType = Interrupt) then */

/* Update the shadow set information for an implementation of */
/* Release 2 of the architecture */
if ((ArchitectureRevision ≥ 2) and (SRSCtlHSS > 0) and (StatusBEV = 0) and
    (StatusERL = 0)) then
    SRSCtlPSS ← SRSCtlCSS
    SRSCtlCSS ← NewShadowSet
endif
endif /* if StatusEXL = 1 then */

CauseCE ← FaultingCoproprocessorNumber
CauseExcCode ← ExceptionType
StatusEXL ← 1

```



```

/* Calculate the vector base address */
if Status_BEV = 1 then
    vectorBase ← 16#BFC0.0200
else
    if ArchitectureRevision ≥ 2 then
        /* The fixed value of EBase31..30 forces the base to be in kseg0 or kseg1 */
        vectorBase ← EBase31..12 || 16#000
    else
        vectorBase ← 16#8000.0000
    endif
endif

/* Exception PC is the sum of vectorBase and vectorOffset */
PC ← vectorBase31..30 || (vectorBase29..0 + vectorOffset29..0)
/* No carry between bits 29 and 30 */

```

## 4.7 Debug Exception Processing

All debug exceptions have the same basic processing flow:

- The *DEPC* register is loaded with the program counter (PC) value at which execution will be restarted and the *DBD* bit is set appropriately in the *Debug* register. The value loaded into the *DEPC* register is the current PC if the instruction is not in the delay slot of a branch, or the PC-4 of the branch if the instruction is in the delay slot of a branch.
- The *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB*, *DINT*, *DIBImpr*, *DDBLImpr*, and *DDBSImpr* bits in the *Debug* register are updated appropriately depending on the debug exception type.
- The *Debug2* register is updated with additional information for complex breakpoints.
- *Halt* and *Doze* bits in the *Debug* register are updated appropriately.
- *DM* bit in the *Debug* register is set to 1.
- The processor is started at the debug exception vector.

The value loaded into *DEPC* represents the restart address for the debug exception and need not be modified by the debug exception handler software in the usual case. Debug software need not look at the *DBD* bit in the *Debug* register unless it wishes to identify the address of the instruction that actually caused the debug exception.

A unique debug exception is indicated through the *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB*, *DINT*, *DIBImpr*, *DDBLImpr*, and *DDBSImpr* bits in the *Debug* register.

No other CP0 registers or fields are changed due to the debug exception, thus no additional state is saved.

### Operation:

```

if InstructionInBranchDelaySlot then
    DEPC ← PC-4
    Debug_DBD ← 1
else
    DEPC ← PC
    Debug_DBD ← 0
endif

```

```

DebugD* bits ← DebugExceptionType
DebugHalt ← HaltStatusAtDebugException
DebugDoze ← DozeStatusAtDebugException
DebugDM ← 1
if EJTAGControlRegisterProbTrap = 1 then
    PC ← 0xFF20_0200
else
    PC ← 0xBFC0_0480
endif

```

The same debug exception vector location is used for all debug exceptions. The location is determined by the Prob-Trap bit in the EJTAG Control register (ECR), as shown in [Table 4.9](#).

**Table 4.9 Debug Exception Vector Addresses**

ProbTrap bit in ECR Register	Debug Exception Vector Address
0	0xBFC0_0480
1	0xFF20_0200 in dmseg

## 4.8 Exception Descriptions

The following subsections describe each of the exceptions listed in the same sequence as shown in [Table 4.1](#).

### 4.8.1 Reset/SoftReset Exception

A reset exception occurs when the *SI\_ColdReset* signal is asserted to the processor; a soft reset occurs when the *SI\_Reset* signal is asserted. These exceptions are not maskable. When one of these exceptions occurs, the processor performs a full reset initialization, including aborting state machines, establishing critical state, and generally placing the processor in a state in which it can execute instructions from uncached, unmapped address space. On a Reset/Soft-Reset exception, the state of the processor is not defined, with the following exceptions:

- The *Config* register is initialized with its boot state.
- The *RP*, *BEV*, *TS*, *SR*, *NMI*, and *ERL* fields of the *Status* register are initialized to a specified state.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC. Note that this value may or may not be predictable.
- PC is loaded with 0xBFC0\_0000.

#### Cause Register ExcCode Value:

None

#### Additional State Saved:

None

#### Entry Vector Used:

Reset (0xBFC0\_0000)

**Operation:**

```

Config ← ConfigurationState
StatusRP ← 0
StatusBEV ← 1
StatusTS ← 0
StatusSR ← 0/1 (depending on Reset or SoftReset)
StatusNMI ← 0
StatusERL ← 1
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else
    ErrorEPC ← PC
endif
PC ← 0xBFC0_0000

```

**4.8.2 Debug Single Step Exception**

A debug single step exception occurs after the CPU has executed one/two instructions in non-debug mode, when returning to non-debug mode after debug mode. One instruction is allowed to execute when returning to a non jump/branch instruction, otherwise two instructions are allowed to execute since the jump/branch and the instruction in the delay slot are executed as one step. Debug single step exceptions are enabled by the SSt bit in the Debug register, and are always disabled for the first one/two instructions after a DERET.

The DEPC register points to the instruction on which the debug single step exception occurred, which is also the next instruction to single step or execute when returning from debug mode. So the DEPC will not point to the instruction which has just been single stepped, but rather the following instruction. The DBD bit in the Debug register is never set for a debug single step exception, since the jump/branch and the instruction in the delay slot is executed in one step.

Exceptions occurring on the instruction(s) executed with debug single step exception enabled are taken even though debug single step was enabled. For a normal exception (other than reset), a debug single step exception is then taken on the first instruction in the normal exception handler. Debug exceptions are unaffected by single step mode, e.g. returning to a SDBBP instruction with debug single step exceptions enabled causes a debug software breakpoint exception, and *DEPC* points to the SDBBP instruction. However, returning to an instruction (not jump/branch) just before the SDBBP instruction, causes a debug single step exception with the *DEPC* pointing to the SDBBP instruction.

To ensure proper functionality of single step, the debug single step exception has priority over all other exceptions, except reset and soft reset.

**Debug Register Debug Status Bit Set**

DSS

**Additional State Saved**

None

**Entry Vector Used**

Debug exception vector

### 4.8.3 Debug Interrupt Exception

A debug interrupt exception is either caused by the *EjtagBrk* bit in the *EJTAG Control* register (controlled through the TAP), or caused by the debug interrupt request signal to the CPU.

The debug interrupt exception is an asynchronous debug exception which is taken as soon as possible, but with no specific relation to the executed instructions. The *DEPC* register is set to the instruction where execution should continue after the debug handler is through. The *DBD* bit is set based on whether the interrupted instruction was executing in the delay slot of a branch.

#### Debug Register Debug Status Bit Set

DINT

#### Additional State Saved

None

#### Entry Vector Used

Debug exception vector

### 4.8.4 Non-Maskable Interrupt (NMI) Exception

A non maskable interrupt exception occurs when the *SI\_NMI* signal is asserted to the processor. *SI\_NMI* is an edge sensitive signal - only one NMI exception will be taken each time it is asserted. An NMI exception occurs only at instruction boundaries, so it does not cause any reset or other hardware initialization. The state of the cache, memory, and other processor states are consistent and all registers are preserved, with the following exceptions:

- The *BEV*, *TS*, *SR*, *NMI*, and *ERL* fields of the *Status* register are initialized to a specified state.
- The *ErrorEPC* register is loaded with PC-4 if the state of the processor indicates that it was executing an instruction in the delay slot of a branch. Otherwise, the *ErrorEPC* register is loaded with PC.
- PC is loaded with 0xBFC0\_0000.

#### Cause Register ExcCode Value:

None

#### Additional State Saved:

None

#### Entry Vector Used:

Reset (0xBFC0\_0000)

#### Operation:

```

StatusBEV ← 1
StatusTS ← 0
StatusSR ← 0
StatusNMI ← 1
StatusERL ← 1
if InstructionInBranchDelaySlot then
    ErrorEPC ← PC - 4
else

```

```

        ErrorEPC ← PC
    endif
    PC ← 0xBFC0_0000

```

## 4.8.5 Interrupt Exception

The interrupt exception occurs when one or more of the six hardware, two software, or timer interrupt requests is enabled by the *Status* register, and the interrupt input is asserted. See 4.3 “Interrupts” on page 65 for more details about the processing of interrupts.

### Register ExcCode Value:

Int

### Additional State Saved:

**Table 4.10 Register States an Interrupt Exception**

Register State	Value
<i>CauseIP</i>	indicates the interrupts that are pending.

### Entry Vector Used:

See 4.3.2 “Generation of Exception Vector Offsets for Vectored Interrupts” on page 74 for the entry vector used, depending on the interrupt mode the processor is operating in.

## 4.8.6 Debug Instruction Break Exception

A debug instruction break exception occurs when an instruction hardware breakpoint matches an executed instruction. The *DEPC* register and *DBD* bit in the *Debug* register indicate the instruction that caused the instruction hardware breakpoint to match. This exception can only occur if instruction hardware breakpoints are implemented.

### Debug Register Debug Status Bit Set:

DIB

### Additional State Saved:

None

### Entry Vector Used:

Debug exception vector

## 4.8.7 Address Error Exception — Instruction Fetch/Data Access

An address error exception occurs on an instruction or data access when an attempt is made to execute one of the following:

- Fetch an instruction, load a word, or store a word that is not aligned on a word boundary
- Load or store a halfword that is not aligned on a halfword boundary
- Reference the kernel address space from user mode

Note that in the case of an instruction fetch that is not aligned on a word boundary, PC is updated before the condition is detected. Therefore, both EPC and BadVAddr point to the unaligned instruction address. In the case of a data access the exception is taken if either an unaligned address or an address that was inaccessible in the current processor mode was referenced by a load or store instruction.

### Cause Register ExcCode Value:

ADEL: Reference was a load or an instruction fetch

ADES: Reference was a store

### Additional State Saved:

**Table 4.11 CP0 Register States on an Address Exception Error**

Register State	Value
BadVAddr	failing address

### Entry Vector Used:

General exception vector (offset 0x180)

## 4.8.8 SRAM Parity Error Exception

A SRAM error exception occurs when an instruction or data reference detects a data error. This exception is not maskable. To avoid disturbing the error in the cache array the exception vector is to an unmapped, uncached address. This exception is precise.

### Cause Register ExcCode Value

N/A

### Additional State Saved

**Table 4.12 CP0 Register States on a SRAM Parity Error Exception**

Register State	Value
<i>CacheErr</i>	Error state
<i>ErrorEPC</i>	Restart PC

### Entry Vector Used

Cache error vector (offset 16#100)

## 4.8.9 Bus Error Exception — Instruction Fetch or Data Access

A bus error exception occurs when an instruction or data access makes a bus request and that request terminates in an error. The bus error exception can occur on either an instruction fetch or a data access. Bus error exceptions that occur on an instruction fetch have a higher priority than bus error exceptions that occur on a data access.

Bus errors taken on any external access on the M14K core are always precise.

### Cause Register ExcCode Value:

IBE: Error on an instruction reference

DBE: Error on a data reference

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

#### 4.8.10 Protection Exception

The protection exception occurs when an access to memory that has been protected by the Memory Protection Unit has been attempted. Or under certain circumstances, attempted write to the EBase register. See the "Security Features of the M14K™ Processor Family" (MD00896) for more information.

**Register ExcCode Value:**

Prot (Cause Code 29)

**Additional State Saved:**

MPU Config Register, Triggered Field

MPU StatusN Register, Cause\* Fields

**Entry Vector Used**

General exception vector (offset 0x180)

#### 4.8.11 Debug Software Breakpoint Exception

A debug software breakpoint exception occurs when an SDBBP instruction is executed. The *DEPC* register and *DBD* bit in the *Debug* register will indicate the SDBBP instruction that caused the debug exception.

**Debug Register Debug Status Bit Set:**

DBp

**Additional State Saved:**

None

**Entry Vector Used:**

Debug exception vector

#### 4.8.12 Execution Exception — System Call

The system call exception is one of the nine execution exceptions. All of these exceptions have the same priority. A system call exception occurs when a SYSCALL instruction is executed.

**Cause Register ExcCode Value:**

Sys

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

#### 4.8.13 Execution Exception — Breakpoint

The breakpoint exception is one of the nine execution exceptions. All of these exceptions have the same priority. A breakpoint exception occurs when a BREAK instruction is executed.

**Cause Register ExcCode Value:**

Bp

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

#### 4.8.14 Execution Exception — Reserved Instruction

The reserved instruction exception is one of the nine execution exceptions. All of these exceptions have the same priority. A reserved instruction exception occurs when a reserved or undefined major opcode or function field is executed. This includes Coprocessor 2 instructions which are decoded reserved in the Coprocessor 2.

**Cause Register ExcCode Value:**

RI

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

#### 4.8.15 Execution Exception — Coprocessor Unusable

The coprocessor unusable exception is one of the nine execution exceptions. All of these exceptions have the same priority. A coprocessor unusable exception occurs when an attempt is made to execute a coprocessor instruction for one of the following:

- a corresponding coprocessor unit that has not been marked usable by setting its CU bit in the *Status* register
- CP0 instructions, when the unit has not been marked usable, and the processor is executing in user mode

**Cause Register ExcCode Value:**

CpU



**Additional State Saved:****Table 4.13 Register States on a Coprocessor Unusable Exception**

Register State	Value
Cause <sub>CE</sub>	Unit number of the coprocessor being referenced

**Entry Vector Used:**

General exception vector (offset 0x180)

**4.8.16 Execution Exception — CorExtend Unusable**

The CorExtend unusable exception is one of the nine execution exceptions. All of these exceptions have the same priority. A CorExtend Unusable exception occurs when an attempt is made to execute a CorExtend instruction when *Status<sub>CEE</sub>* is cleared. It is implementation dependent whether this functionality is supported. Generally, the functionality will only be supported if a CorExtend block contains local destination registers

**Cause Register ExcCode Value:**

CEU

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

**4.8.17 Execution Exception — Coprocessor 2 Exception**

The Coprocessor 2 exception is one of the nine execution exceptions. All of these exceptions have the same priority. A Coprocessor 2 exception occurs when a valid Coprocessor 2 instruction cause a general exception in the Coprocessor 2.

**Cause Register ExcCode Value:**

C2E

**Additional State Saved:**

Depending on the Coprocessor 2 implementation, additional state information of the exception can be saved in a Coprocessor 2 control register.

**Entry Vector Used:**

General exception vector (offset 0x180)

**4.8.18 Execution Exception — Implementation-Specific 1 Exception**

The Implementation-Specific 1 exception is one of the nine execution exceptions. All of these exceptions have the same priority. An implementation-specific 1 exception occurs when a valid coprocessor 2 instruction cause an implementation-specific 1 exception in the Coprocessor 2.

**Cause Register ExcCode Value:**

IS1

**Additional State Saved:**

Depending on the coprocessor 2 implementation, additional state information of the exception can be saved in a coprocessor 2 control register.

**Entry Vector Used:**

General exception vector (offset 0x180)

### 4.8.19 Execution Exception — Integer Overflow

The integer overflow exception is one of the nine execution exceptions. All of these exceptions have the same priority. An integer overflow exception occurs when selected integer instructions result in a 2's complement overflow.

**Cause Register ExcCode Value:**

Ov

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

### 4.8.20 Execution Exception — Trap

The trap exception is one of the nine execution exceptions. All of these exceptions have the same priority. A trap exception occurs when a trap instruction results in a TRUE value.

**Cause Register ExcCode Value:**

Tr

**Additional State Saved:**

None

**Entry Vector Used:**

General exception vector (offset 0x180)

### 4.8.21 Debug Data Break Exception

A debug data break exception occurs when a data hardware breakpoint matches the load/store transaction of an executed load/store instruction. The *DEPC* register and DBD bit in the *Debug* register will indicate the load/store instruction that caused the data hardware breakpoint to match. The load/store instruction that caused the debug exception has not completed e.g. not updated the register file, and the instruction can be re-executed after returning from the debug handler.

**Debug Register Debug Status Bit Set:**

DDBL for a load instruction or DDBS for a store instruction

**Additional State Saved:**

None

**Entry Vector Used:**

Debug exception vector

**4.8.22 Complex Break Exception**

A complex data break exception occurs when the complex hardware breakpoint detects an enabled breakpoint. Complex breaks are taken imprecisely—the instruction that actually caused the exception is allowed to complete and the *DEPC* register and DBD bit in the *Debug* register point to a following instruction.

**Debug Register Debug Status Bit Set:***DIBImpr*, *DDBLImpr*, and/or *DDBSImpr***Additional State Saved:***Debug2* fields indicate which type(s) of complex breakpoints were detected.**Entry Vector Used:**

Debug exception vector

**4.9 Exception Handling and Servicing Flowcharts**

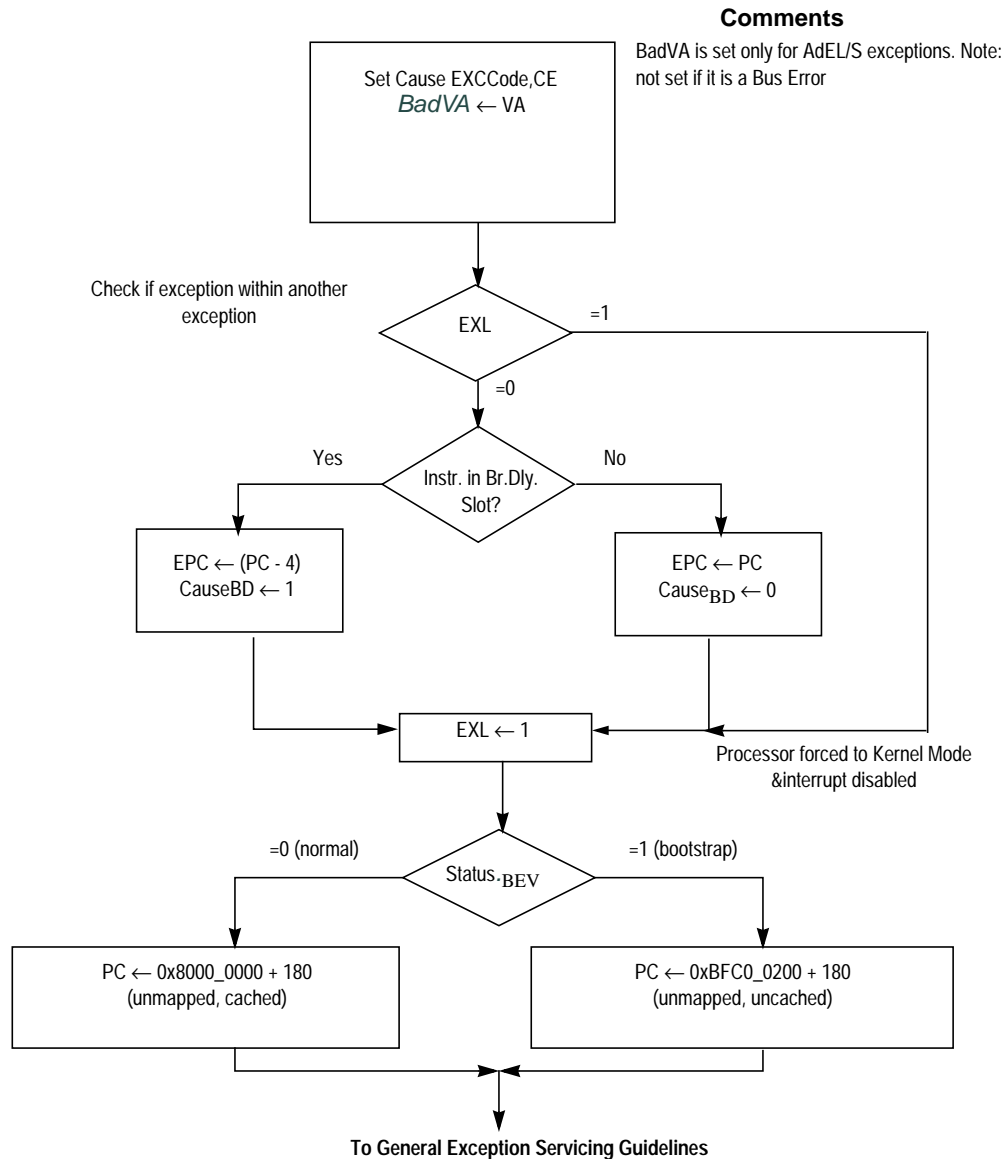
The remainder of this chapter contains flowcharts for the following exceptions and guidelines for their handlers:

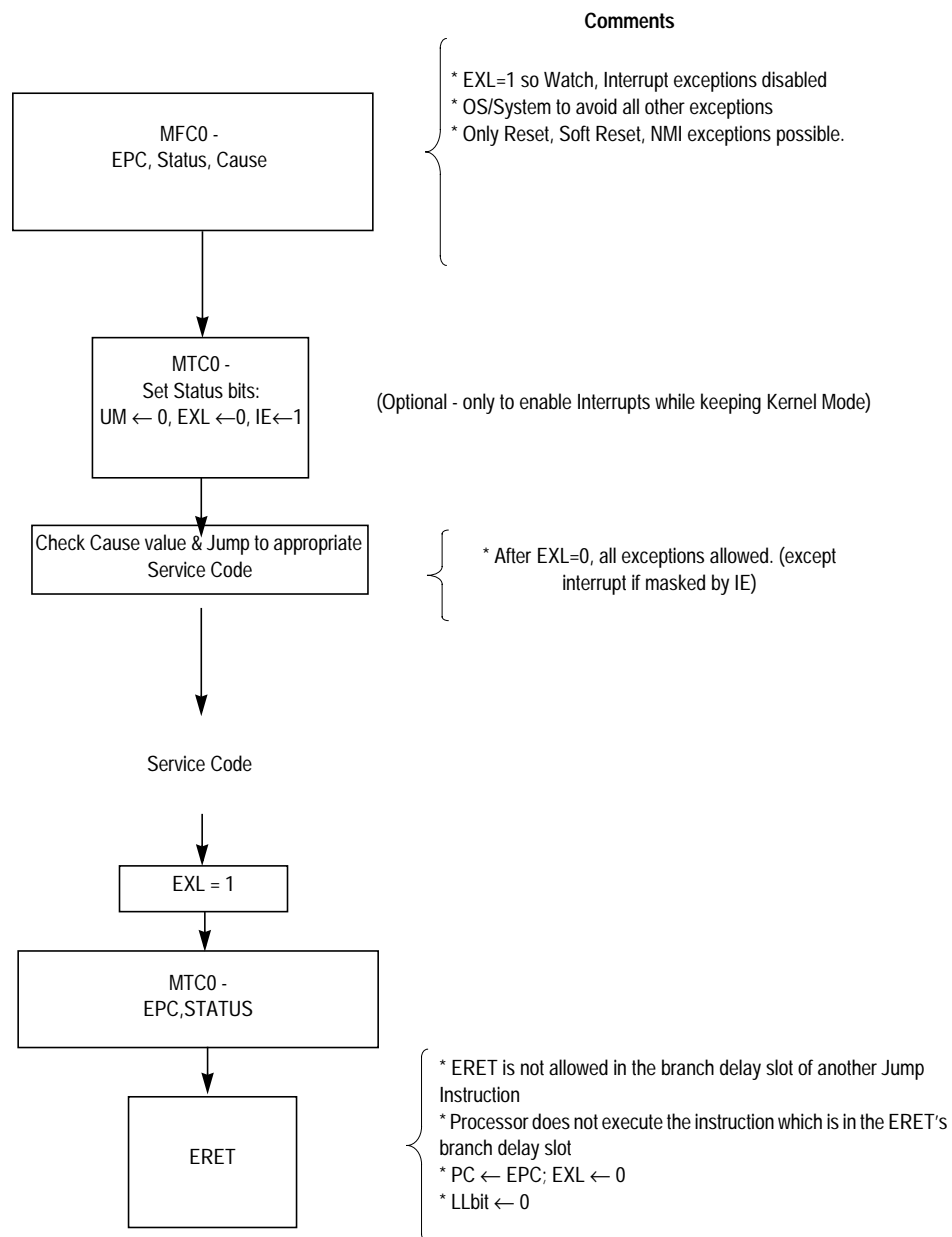
- General exceptions and their exception handler
- Reset, soft reset and NMI exceptions, and a guideline to their handler
- Debug exceptions

**Figure 4.3 General Exception Handler (HW)**

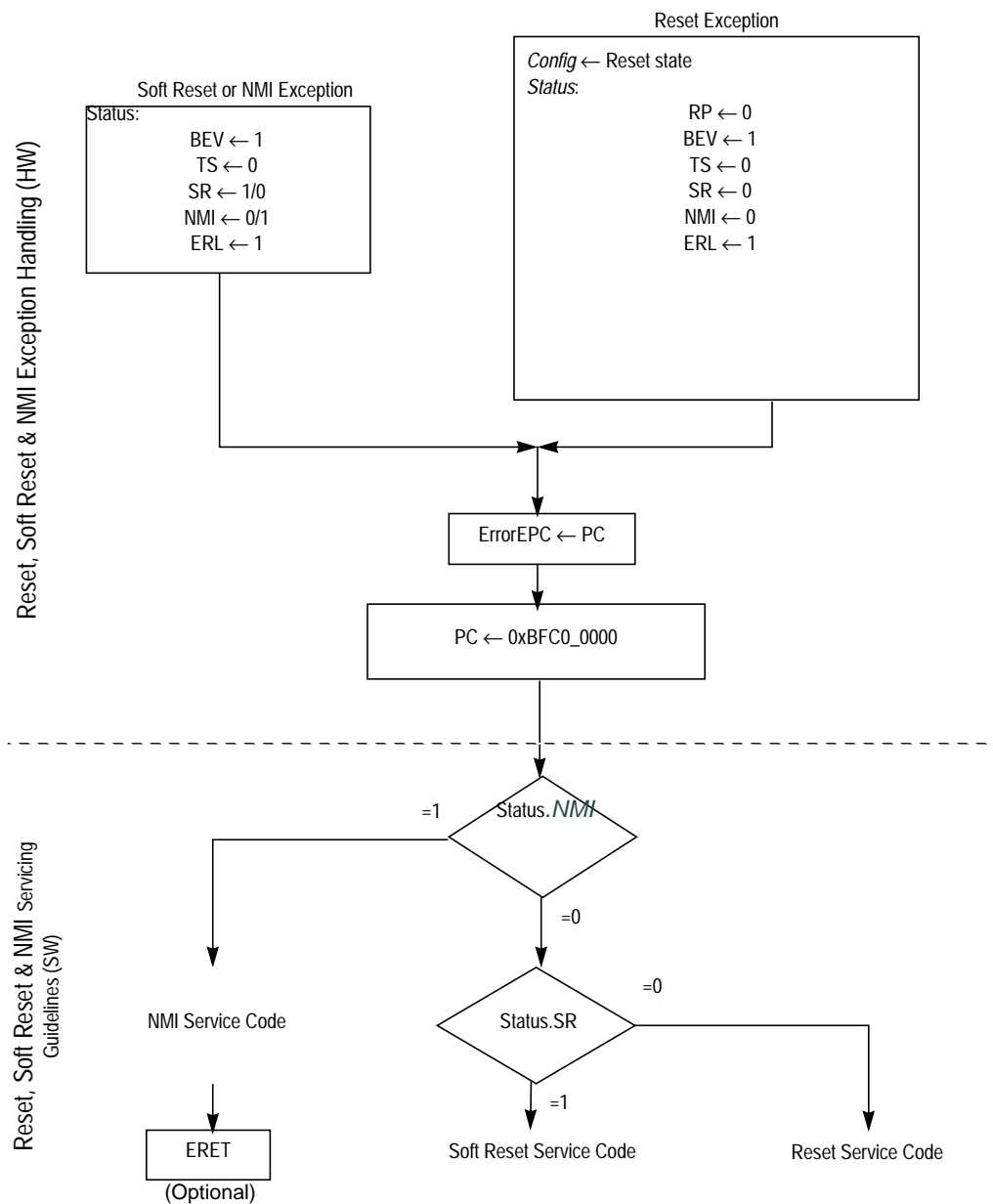
Exceptions other than Reset, Soft Reset, NMI, EJTag Debug and cache error, or first-level TLB miss.

Note: Interrupts can be masked by IE or IMs and Watch is masked if EXL = 1



**Figure 4.4 General Exception Servicing Guidelines (SW)**

**Figure 4.5 Reset, Soft Reset and NMI Exception Handling and Servicing Guidelines**



## CP0 Registers of the M14K™ Core

The System Control Coprocessor (CP0) provides the register interface to the M14K processor core and supports memory management, address translation, exception handling, and other privileged operations. Each CP0 register has a unique number that identifies it; this number is referred to as the *register number*. For instance, the *PageMask* register is register number 5. For more information on the EJTAG registers, refer to [Chapter 4, “Exceptions and Interrupts in the M14K™ Core”](#) on page 63.

After updating a CP0 register there is a hazard period of zero or more instructions from the update instruction (MTC0) and until the effect of the update has taken place in the core. Refer to [Chapter 10, “M14K™ Processor Core Instructions”](#) on page 239 for further details on CP0 hazards.

The current chapter contains the following sections:

- [Section 5.1 “CP0 Register Summary”](#)
- [Section 5.2 “CP0 Register Descriptions”](#)

### 5.1 CP0 Register Summary

[Table 5.1](#) lists the CP0 registers in numerical order. The individual registers are described throughout this chapter. Where more than one registers shares the same register number at different values of the “sel” field of the instruction, their names are listed using a slash (/) as separator.

**Table 5.1 CP0 Registers**

Register Number	Register Name	Function
0-3	Reserved	Reserved in the M14K core.
4	<i>UserLocal</i>	User information that can be written by privileged software and read via <i>RDHWR</i> register 29.
5-6	Reserved	Reserved in the M14K core.
7	<i>HWREna</i>	Enables access via the <i>RDHWR</i> instruction to selected hardware registers in non-privileged mode.
8	<i>BadVAddr</i> <sup>1</sup>	Reports the address for the most recent address-related exception.
9	<i>Count</i> <sup>1</sup>	Processor cycle count.
10	Reserved	Reserved in the M14K core.
11	<i>Compare</i> <sup>1</sup>	Timer interrupt control.

Table 5.1 CP0 Registers (Continued)

Register Number	Register Name	Function
12	<i>Status</i> <i>IntCtl</i> <i>SRSCtl</i> <i>SRSMap1</i> <i>View_IPL</i> <i>SRSMap2</i>	Processor status and control; interrupt control; and shadow set control.
13	<i>Cause</i> <sup>1</sup> <i>View_RIPL</i> <i>NestedExc</i>	Cause of last exception.
14	<i>EPC</i> <sup>1</sup> <i>NestedEPC</i>	Program counter at last exception.
15	<i>PRId</i> / <i>EBase</i> <i>CDMMBase</i>	Processor identification and revision; exception base address; Common Device Memory Map Base register.
16	<i>Config</i> / <i>Config1</i> <i>Config2</i> <i>Config3</i> <i>Config4</i> <i>Config5</i> <i>Config7</i>	Configuration registers.
17-22	Reserved	Reserved in the M14K core.
23	<i>Debug</i> / <i>Debug2</i> / <i>TraceControl</i> <i>TraceControl2</i> <i>UserTraceData1</i> <i>TraceBPC</i> <sup>2</sup>	EJTAG Debug register EJTAG Debug register 2 EJTAG Trace Control register EJTAG Trace Control register2 EJTAG User Trace Data1 register EJTAG Trace Breakpoint Register
24	<i>DEPC</i> <sup>2</sup> <i>UserTraceData2</i>	Program counter at last debug exception. EJTAG User Trace Data2 register
25	<i>PerfCtl0</i> <i>PerfCnt0</i> <i>PerfCtl1</i> <i>PerfCnt1</i>	Performance counter 0 control Performance counter 0 Performance counter 1control Performance counter 1
26	<i>ErrCtl</i>	Software parity check enable.
27	<i>CacheErr</i>	Records information about SRAM parity errors.
28-29	Reserved	Reserved in the M14K core.
30	<i>ErrorEPC</i> <sup>1</sup>	Program counter at last error.
31	<i>DeSAVE</i> <sup>2</sup>	Debug handler scratchpad register.
1. Registers used in exception processing. 2. Registers used in debug.		



## 5.2 CP0 Register Descriptions

The CP0 registers provide the interface between the ISA and the architecture. Each register is described below, with the registers presented in numerical order, first by register number, then by select field number.

For each register described below, field descriptions include the read/write properties of the field and the reset state of the field. The read/write access properties of the field are described in [Table 5.2](#).

**Table 5.2 CP0 Register Field Types**

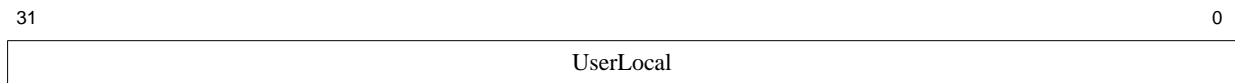
Read/Write Notation	Hardware Interpretation	Software Interpretation
R/W	A field in which all bits are readable and writable by software and, potentially, by hardware. Hardware updates of this field are visible by software reads. Software updates of this field are visible by hardware reads. If the reset state of this field is “Undefined,” either software or hardware must initialize the value before the first read will return a predictable value. This should not be confused with the formal definition of UNDEFINED behavior.	
R	A field that is either static or is updated only by hardware. If the Reset State of this field is either “0” or “Preset”, hardware initializes this field to zero or to the appropriate state, respectively, on powerup. If the Reset State of this field is “Undefined”, hardware updates this field only under those conditions specified in the description of the field.	A field to which the value written by software is ignored by hardware. Software may write any value to this field without affecting hardware behavior. Software reads of this field return the last value updated by hardware. If the Reset State of this field is “Undefined,” software reads of this field result in an UNPREDICTABLE value except after a hardware update done under the conditions specified in the description of the field.
W	A field that can be written by software but which can not be read by software. Software reads of this field will return an UNDEFINED value.	
0	A field that hardware does not update, and for which hardware can assume a zero value.	A field to which the value written by software must be zero. Software writes of non-zero values to this field may result in UNDEFINED behavior of the hardware. Software reads of this field return zero as long as all previous software writes are zero. If the Reset State of this field is “Undefined,” software must write this field with zero before it is guaranteed to read as zero.

### 5.2.1 UserLocal Register (CP0 Register 4, Select 2)

The *UserLocal* register is a read-write register that is not interpreted by the hardware and conditionally readable via the RDHWR instruction.

[Figure 5.1](#) shows the format of the *UserLocal* register; [Table 5.3](#) describes the *UserLocal* register fields.

**Figure 5.1 UserLocal Register Format**



### Table 5.3 UserLocal Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bit(s)			
UserLocal	31:0	This field contains software information that is not interpreted by hardware.	R/W	Undefined

## Programming Notes

Privileged software may write this register with arbitrary information and make it accessible to unprivileged software via register 29 (*ULR*) of the RDHWR instruction. To do so, bit 29 of the *HWREna* register must be set to a 1 to enable unprivileged access to the register. In some operating environments, the *UserLocal* register contains a pointer to a thread-specific storage block that is obtained via the *RDHWR* register.

### 5.2.2 HWREna Register (CP0 Register 7, Select 0)

The *HWREna* register contains a bit mask that determines which hardware registers are accessible via the RDHWR instruction.

Figure 5.2 shows the format of the *HWREna* Register; Table 5.4 describes the *HWREna* register fields.

### Figure 5.2 HWREna Register Format

31	30	29	28																	4	3	0
0	ULR	0																Mask				

### Table 5.4 HWREna Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:30	Must be written with zero; returns zero on read	0	0
0	28:4	Must be written with zero; returns zero on read	0	0
ULR	29	User Local Register. This register provides read access to the coprocessor 0 <i>UserLocal</i> register. In some operating environments, the <i>UserLocal</i> register is a pointer to a thread-specific storage block.	R/W	0
Mask	3:0	Each bit in this field enables access by the RDHWR instruction to a particular hardware register (which may not be an actual register). If bit ‘n’ in this field is a 1, access is enabled to hardware register ‘n’. If bit ‘n’ of this field is a 0, access is disabled. See the RDHWR instruction for a list of valid hardware registers.	R/W	0

Privileged software may determine which of the hardware registers are accessible by the RDHWR instruction. In doing so, a register may be virtualized at the cost of handling a Reserved Instruction Exception, interpreting the

instruction, and returning the virtualized value. For example, if it is not desirable to provide direct access to the *Count* register, access to that register may be individually disabled and the return value can be virtualized by the operating system.

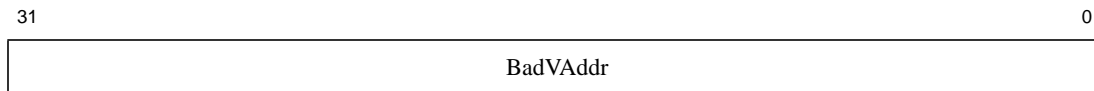
### 5.2.3 BadVAddr Register (CP0 Register 8, Select 0)

The *BadVAddr* register is a read-only register that captures the most recent virtual address that caused the following exception:

- Address error (AdEL or AdES)

The *BadVAddr* register does not capture address information for bus errors, because they are not addressing errors.

**Figure 5.3 BadVAddr Register Format**



**Table 5.5 BadVAddr Register Field Description**

Fields		Description	Read/Write	Reset State
Name	Bits			
BadVAddr	31:0	Bad virtual address.	R	Undefined

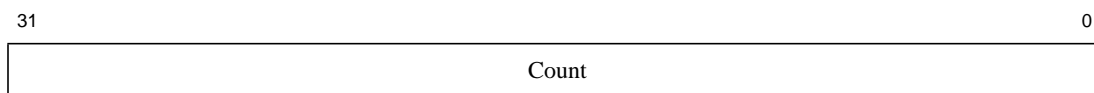
### 5.2.4 Count Register (CP0 Register 9, Select 0)

The *Count* register acts as a timer, incrementing at a constant rate, whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. The counter increments every other clock, if the *DC* bit in the *Cause* register is 0.

The *Count* register can be written for functional or diagnostic purposes, including at reset or to synchronize processors.

By writing the *CountDM* bit in the *Debug* register, it is possible to control whether the *Count* register continues incrementing while the processor is in debug mode.

**Figure 5.4 Count Register Format**



**Table 5.6 Count Register Field Description**

Fields		Description	Read/Write	Reset State
Name	Bits			
Count	31:0	Interval counter.	R/W	Undefined

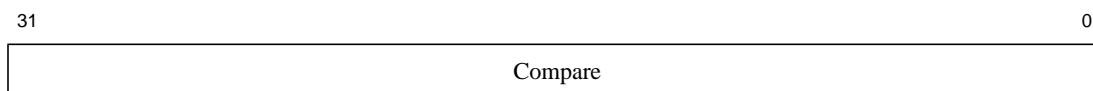
### 5.2.5 Compare Register (CP0 Register 11, Select 0)

The *Compare* register acts in conjunction with the *Count* register to implement a timer and timer interrupt function. The timer interrupt is an output of the cores. The *Compare* register maintains a stable value and does not change on its own.

When the value of the *Count* register equals the value of the *Compare* register, the *SL\_TimerInt* pin is asserted. This pin will remain asserted until the *Compare* register is written. The *SL\_TimerInt* pin can be fed back into the core on one of the interrupt pins to generate an interrupt. Traditionally, this has been done by multiplexing it with hardware interrupt 5 to set interrupt bit *IP*(7) in the *Cause* register.

For diagnostic purposes, the *Compare* register is a read/write register. In normal use, however, the *Compare* register is write-only. Writing a value to the *Compare* register, as a side effect, clears the timer interrupt.

**Figure 5.5 Compare Register Format**



**Table 5.7 Compare Register Field Description**

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Compare	31:0	Interval count compare value.	R/W	Undefined

### 5.2.6 Status Register (CP0 Register 12, Select 0)

The *Status* register is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. Fields of this register combine to create operating modes for the processor. Refer to [3.2 “Modes of Operation” on page 51](#) for a discussion of operating modes, and [4.3 “Interrupts” on page 65](#) for a discussion of interrupt modes.

**Interrupt Enable:** Interrupts are enabled when all of the following conditions are true:

- $IE = 1$
- $EXL = 0$
- $ERL = 0$
- $DM = 0$

If these conditions are met, then the settings of the *IM* and *IE* bits enable the interrupts.

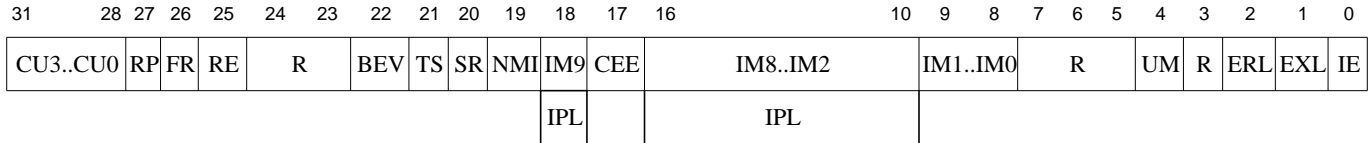
**Operating Modes:** If the *DM* bit in the *Debug* register is 1, then the processor is in debug mode; otherwise the processor is in either kernel or user mode. The following CPU *Status* register bit settings determine user or kernel mode:

- User mode:  $UM = 1$ ,  $EXL = 0$ , and  $ERL = 0$
- Kernel mode:  $UM = 0$ , or  $EXL = 1$ , or  $ERL = 1$

Coprocessor Accessibility: The *Status* register *CU* bits control coprocessor accessibility. If any coprocessor is unusable, then an instruction that accesses it generates an exception.

Figure 5.6 shows the format of the *Status* register; Table 5.8 describes the *Status* register fields.

**Figure 5.6 Status Register Format**



**Table 5.8 Status Register Field Descriptions**

Fields		Description	Read/Write	Reset State						
Name	Bits									
CU3	31	Controls access to coprocessor 3. COP3 is not supported. This bit cannot be written and will read as 0.	R	0						
CU2	30	Controls access to coprocessor 2. This bit can only be written if coprocessor is attached to the COP2 interface. ( <i>C2</i> bit in Config1 is set). This bit will read as 0 if no coprocessor is present.	R/W	0						
CU1	29	Controls access to Coprocessor 1. COP1 is not supported. This bit cannot be written and will read as 0.	R	0						
CU0	28	Controls access to coprocessor 0: <table border="1"><thead><tr><th>Encoding</th><th>Meaning</th></tr></thead><tbody><tr><td>0</td><td>Access not allowed</td></tr><tr><td>1</td><td>Access allowed</td></tr></tbody></table> Coprocessor 0 is always usable when the processor is running in kernel mode, independent of the state of the <i>CU0</i> bit.	Encoding	Meaning	0	Access not allowed	1	Access allowed	R/W	Undefined
Encoding	Meaning									
0	Access not allowed									
1	Access allowed									
RP	27	Enables reduced power mode. The state of the <i>RP</i> bit is available on the external core interface as the <i>SI_RP</i> signal.	R/W	0 for Cold Reset only.						
FR	26	This bit is related to floating point registers. Because the M14K core does not contain a floating point unit, this bit is ignored on writes and reads as zero.	R	0						
RE	25	Used to enable reverse-endian memory references while the processor is running in user mode: <table border="1"><thead><tr><th>Encoding</th><th>Meaning</th></tr></thead><tbody><tr><td>0</td><td>User mode uses configured endianness</td></tr><tr><td>1</td><td>User mode uses reversed endianness</td></tr></tbody></table> Neither Debug Mode nor Kernel Mode nor Supervisor Mode references are affected by the state of this bit.	Encoding	Meaning	0	User mode uses configured endianness	1	User mode uses reversed endianness	R/W	Undefined
Encoding	Meaning									
0	User mode uses configured endianness									
1	User mode uses reversed endianness									

Table 5.8 Status Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State						
Name	Bits									
R	24:23	Reserved. This field is ignored on writes and reads as 0.	R	0						
BEV	22	Controls the location of exception vectors: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Normal</td></tr><tr><td>1</td><td>Bootstrap</td></tr></table>	Encoding	Meaning	0	Normal	1	Bootstrap	R/W	1
Encoding	Meaning									
0	Normal									
1	Bootstrap									
TS	21	TLB shutdown.Because the M14K core does not contain a TLB, this bit is ignored on writes and reads as 0.	R	0						
SR	20	Indicates that the entry through the reset exception vector was due to a Soft Reset: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Not Soft Reset (NMI or Reset)</td></tr><tr><td>1</td><td>Soft Reset</td></tr></table> <p>Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.</p>	Encoding	Meaning	0	Not Soft Reset (NMI or Reset)	1	Soft Reset	R/W	1 for Soft Reset; 0 otherwise
Encoding	Meaning									
0	Not Soft Reset (NMI or Reset)									
1	Soft Reset									
NMI	19	Indicates that the entry through the reset exception vector was due to an NMI: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Not NMI (Soft Reset or Reset)</td></tr><tr><td>1</td><td>NMI</td></tr></table> <p>Software can only write a 0 to this bit to clear it and cannot force a 0-1 transition.</p>	Encoding	Meaning	0	Not NMI (Soft Reset or Reset)	1	NMI	R/W	1 for NMI; 0 otherwise
Encoding	Meaning									
0	Not NMI (Soft Reset or Reset)									
1	NMI									
CEE	17	CorExtend Enable: Implementation dependent. If CorExtend block indicates that this bit should be used, any attempt to execute a CorExtend instruction with this bit cleared will result in a CorExtend Unusable exception. This bit is reserved if CorExtend is not present.	R/W	Undefined						
IM9..IM2	18, 16:10	Interrupt Mask: Controls the enabling of each of the hardware interrupts. Refer to 4.3 “Interrupts” on page 65 for a complete discussion of enabled interrupts. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupt request disabled</td></tr><tr><td>1</td><td>Interrupt request enabled</td></tr></table> <p>In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled (<i>Config3</i><sub>VEIC</sub> = 1), these bits take on a different meaning and are interpreted as the <i>IPL</i> field, described below.</p>	Encoding	Meaning	0	Interrupt request disabled	1	Interrupt request enabled	R/W	Undefined for IM7:IM2  0 for IM9:IM8
Encoding	Meaning									
0	Interrupt request disabled									
1	Interrupt request enabled									

Table 5.8 Status Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State						
Name	Bits									
IPL	18, 16:10	Interrupt Priority Level. In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled ( <i>Config3</i> <sub>VEIC</sub> = 1), this field is the encoded (0..63) value of the current IPL. An interrupt will be signaled only if the requested IPL is higher than this value. If EIC interrupt mode is not enabled ( <i>Config3</i> <sub>VEIC</sub> = 0), these bits take on a different meaning and are interpreted as the <i>IM7..IM2</i> bits, described above.	R/W	Undefined for IPL15:IPL10  0 for IPL18:IPL17						
IM1..IM0	9:8	Interrupt Mask: Controls the enabling of each of the software interrupts. Refer to <a href="#">Section 4.3 “Interrupts”</a> for a complete discussion of enabled interrupts. <table border="1"><thead><tr><th>Encoding</th><th>Meaning</th></tr></thead><tbody><tr><td>0</td><td>Interrupt request disabled</td></tr><tr><td>1</td><td>Interrupt request enabled</td></tr></tbody></table> In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled, these bits are writable, but have no effect on the interrupt system.	Encoding	Meaning	0	Interrupt request disabled	1	Interrupt request enabled	R/W	Undefined
Encoding	Meaning									
0	Interrupt request disabled									
1	Interrupt request enabled									
R	7:5	Reserved. This field is ignored on writes and reads as 0.	R	0						
UM	4	This bit denotes the base operating mode of the processor. See <a href="#">Section 3.2 “Modes of Operation”</a> for a full discussion of operating modes. The encoding of this bit is: <table border="1"><thead><tr><th>Encoding</th><th>Meaning</th></tr></thead><tbody><tr><td>0</td><td>Base mode is Kernel Mode</td></tr><tr><td>1</td><td>Base mode is User Mode</td></tr></tbody></table> Note that the processor can also be in kernel mode if <i>ERL</i> or <i>EXL</i> is set, regardless of the state of the <i>UM</i> bit.	Encoding	Meaning	0	Base mode is Kernel Mode	1	Base mode is User Mode	R/W	Undefined
Encoding	Meaning									
0	Base mode is Kernel Mode									
1	Base mode is User Mode									
R	3	This bit is reserved. This bit is ignored on writes and reads as zero.	R	0						

Table 5.8 Status Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State						
Name	Bits									
ERL	2	<p>Error Level; Set by the processor when a Reset, Soft Reset, NMI or Cache Error exception are taken.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Normal level</td></tr><tr><td>1</td><td>Error level</td></tr></table> <p>When <i>ERL</i> is set:</p> <ul style="list-style-type: none"><li>• The processor is running in kernel mode</li><li>• Interrupts are disabled</li><li>• The ERET instruction will use the return address held in <i>ErrorEPC</i> instead of <i>EPC</i></li><li>• The lower 2<sup>29</sup> bytes of kuseg are treated as an unmapped and uncached region. See <a href="#">Chapter 3, “Memory Management of the M14K™ Core” on page 51</a>. This allows main memory to be accessed in the presence of cache errors. The operation of the processor is <b>UNDEFINED</b> if the <i>ERL</i> bit is set while the processor is executing instructions from kuseg.</li></ul>	Encoding	Meaning	0	Normal level	1	Error level	R/W	1
Encoding	Meaning									
0	Normal level									
1	Error level									
EXL	1	<p>Exception Level; Set by the processor when any exception other than Reset, Soft Reset, or NMI exceptions is taken.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Normal level</td></tr><tr><td>1</td><td>Exception level</td></tr></table> <p>When <i>EXL</i> is set:</p> <ul style="list-style-type: none"><li>• The processor is running in Kernel Mode</li><li>• Interrupts are disabled.</li><li>• <i>EPC</i>, <i>Cause</i><sub>BD</sub> and <i>SRSCtl</i> (implementations of Release 2 of the Architecture only) will not be updated if another exception is taken</li></ul>	Encoding	Meaning	0	Normal level	1	Exception level	R/W	Undefined
Encoding	Meaning									
0	Normal level									
1	Exception level									
IE	0	<p>Interrupt Enable: Acts as the master enable for software and hardware interrupts:</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupts are disabled</td></tr><tr><td>1</td><td>Interrupts are enabled</td></tr></table> <p>In Release 2 of the Architecture, this bit may be modified separately via the DI and EI instructions.</p>	Encoding	Meaning	0	Interrupts are disabled	1	Interrupts are enabled	R/W	Undefined
Encoding	Meaning									
0	Interrupts are disabled									
1	Interrupts are enabled									

### 5.2.7 IntCtl Register (CP0 Register 12, Select 1)

The *IntCtl* register controls the expanded interrupt capability added in Release 2 of the Architecture, including vectored interrupts and support for an external interrupt controller. This register does not exist in implementations of Release 1 of the Architecture.



Figure 5.7 shows the format of the *IntCtl* register; Table 5.9 describes the *IntCtl* register fields.

**Figure 5.7 IntCtl Register Format**

31	29	28	26	25	23	22	21	20	16	15	14	13	12	10	9	5	4	0
IPTI	IPPCI	IPFDC	PF	ICE	StkDec	Clr EXL	APE	Use KStk	000	VS	0							

**Table 5.9 IntCtl Register Field Descriptions**

Fields		Description	Read/Write	Reset State																					
Name	Bits																								
IPTI	31:29	For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Timer Interrupt request is merged, and allows software to determine whether to consider <i>Cause<sub>TI</sub></i> for a potential interrupt.	R	Externally Set																					
		<table><tr><th>Encoding</th><th>IP bit</th><th>Hardware Interrupt Source</th></tr><tr><td>2</td><td>2</td><td>HW0</td></tr><tr><td>3</td><td>3</td><td>HW1</td></tr><tr><td>4</td><td>4</td><td>HW2</td></tr><tr><td>5</td><td>5</td><td>HW3</td></tr><tr><td>6</td><td>6</td><td>HW4</td></tr><tr><td>7</td><td>7</td><td>HW5</td></tr></table>			Encoding	IP bit	Hardware Interrupt Source	2	2	HW0	3	3	HW1	4	4	HW2	5	5	HW3	6	6	HW4	7	7	HW5
		Encoding			IP bit	Hardware Interrupt Source																			
		2			2	HW0																			
		3			3	HW1																			
		4			4	HW2																			
		5			5	HW3																			
		6			6	HW4																			
		7			7	HW5																			
The value of this bit is set by the static input, <i>SI_IPTI[2:0]</i> . This allows external logic to communicate the specific <i>SI_Int</i> hardware interrupt pin to which the <i>SI_TimerInt</i> signal is attached.																									
The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode.																									

Table 5.9 IntCtl Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State																					
Name	Bits																								
IPPCI	28:26	For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Performance Counter Interrupt request is merged, and allows software to determine whether to consider <i>Cause</i> <sub>PCI</sub> for a potential interrupt.	R	0																					
		<table><tr><th>Encoding</th><th>IP bit</th><th>Hardware Interrupt Source</th></tr><tr><td>2</td><td>2</td><td>HW0</td></tr><tr><td>3</td><td>3</td><td>HW1</td></tr><tr><td>4</td><td>4</td><td>HW2</td></tr><tr><td>5</td><td>5</td><td>HW3</td></tr><tr><td>6</td><td>6</td><td>HW4</td></tr><tr><td>7</td><td>7</td><td>HW5</td></tr></table>			Encoding	IP bit	Hardware Interrupt Source	2	2	HW0	3	3	HW1	4	4	HW2	5	5	HW3	6	6	HW4	7	7	HW5
		Encoding			IP bit	Hardware Interrupt Source																			
		2			2	HW0																			
		3			3	HW1																			
		4			4	HW2																			
		5			5	HW3																			
		6			6	HW4																			
		7			7	HW5																			
		The value of this bit is set by the static input, <i>SI_IPPCI[2:0]</i> . This allows external logic to communicate the specific <i>SI_Int</i> hardware interrupt pin to which the <i>SI_PCInt</i> signal is attached.																							
The value of this field is not meaningful if External Interrupt Controller Mode is enabled. The external interrupt controller is expected to provide this information for that interrupt mode.																									
IPFDC	25:23	For Interrupt Compatibility and Vectored Interrupt modes, this field specifies the IP number to which the Fast Debug Channel Interrupt request is merged, and allows software to determine whether to consider <i>Cause</i> <sub>FDC</sub> for a potential interrupt.	R	Preset or Externally Set																					
		<table><tr><th>Encoding</th><th>IP bit</th><th>Hardware Interrupt Source</th></tr><tr><td>2</td><td>2</td><td>HW0</td></tr><tr><td>3</td><td>3</td><td>HW1</td></tr><tr><td>4</td><td>4</td><td>HW2</td></tr><tr><td>5</td><td>5</td><td>HW3</td></tr><tr><td>6</td><td>6</td><td>HW4</td></tr><tr><td>7</td><td>7</td><td>HW5</td></tr></table>			Encoding	IP bit	Hardware Interrupt Source	2	2	HW0	3	3	HW1	4	4	HW2	5	5	HW3	6	6	HW4	7	7	HW5
		Encoding			IP bit	Hardware Interrupt Source																			
		2			2	HW0																			
		3			3	HW1																			
		4			4	HW2																			
		5			5	HW3																			
		6			6	HW4																			
		7			7	HW5																			
		The value of this field is <b>UNPREDICTABLE</b> if External Interrupt Controller Mode is both implemented and enabled. The external interrupt controller is expected to provide this information for that interrupt mode.																							
If EJTAG FDC is not implemented, this field returns zero on read.																									

Table 5.9 IntCtl Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State		
Name	Bits					
PF	22	Enables Vector Prefetching Feature.	RW	0		
		Encoding			Meaning	
		0			Vector Prefetching disabled.	
		1			Vector Prefetching enabled.	
ICE	21	For IRET instruction. Enables Interrupt Chaining.	RW	0		
		Encoding			Meaning	
		0			Interrupt Chaining disabled	
		1			Interrupt Chaining enabled	
StkDec	20:16	For Auto-Prologue feature. This is the number of 4-byte words that is decremented from the value of GPR29	RW	0x3		
		Encoding			Decrement Amount in Words	Decrement Amount in Bytes
		0-3			3	12
		Others			As encoded, e.g. 0x5 means 5 words	4 * encoded value e.g. 0x5 means 20 bytes
ClrEXL	15	For Auto-Prologue feature and IRET instruction. If set, during Auto-Prologue and IRET interrupt chaining, the <i>KSU/ERL/EXL</i> fields are cleared.	RW	0		
		Encoding			Meaning	
		0			Fields are not cleared by these operations.	
		1			Fields are cleared by these operations.	
APE	14	Enables Auto-Prologue feature.	RW	0		
		Encoding			Meaning	
		0			Auto-Prologue disabled	
		1			Auto-Prologue enabled	

Table 5.9 IntCtl Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State																					
Name	Bits																								
UseKStk	13	Chooses which Stack to use during Interrupt Automated Prologue.	RW	0																					
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Copy \$29 of the Previous SRS to the Current SRS at the beginning of IAP.  This is used for Bare-Iron environments with only one stack.</td></tr><tr><td>1</td><td>Use \$29 of the Current SRS at the beginning of IAP. This is used for environments where there are separate User-mode and Kernel mode stacks. In this case, \$29 of the SRS used during IAP must be pre-initialized by software to hold the Kernel mode stack pointer.</td></tr></table>			Encoding	Meaning	0	Copy \$29 of the Previous SRS to the Current SRS at the beginning of IAP.  This is used for Bare-Iron environments with only one stack.	1	Use \$29 of the Current SRS at the beginning of IAP. This is used for environments where there are separate User-mode and Kernel mode stacks. In this case, \$29 of the SRS used during IAP must be pre-initialized by software to hold the Kernel mode stack pointer.															
		Encoding			Meaning																				
		0			Copy \$29 of the Previous SRS to the Current SRS at the beginning of IAP.  This is used for Bare-Iron environments with only one stack.																				
1	Use \$29 of the Current SRS at the beginning of IAP. This is used for environments where there are separate User-mode and Kernel mode stacks. In this case, \$29 of the SRS used during IAP must be pre-initialized by software to hold the Kernel mode stack pointer.																								
0	12:10	Must be written as zero; returns zero on read.	0	0																					
VS	9:5	Vector Spacing. If vectored interrupts are implemented (as denoted by <i>Config3<sub>VInt</sub></i> or <i>Config3<sub>VEIC</sub></i> ), this field specifies the spacing between vectored interrupts.	R/W	0																					
		<table><tr><th>Encoding</th><th>Spacing Between Vectors (hex)</th><th>Spacing Between Vectors (decimal)</th></tr><tr><td>16#00</td><td>16#000</td><td>0</td></tr><tr><td>16#01</td><td>16#020</td><td>32</td></tr><tr><td>16#02</td><td>16#040</td><td>64</td></tr><tr><td>16#04</td><td>16#080</td><td>128</td></tr><tr><td>16#08</td><td>16#100</td><td>256</td></tr><tr><td>16#10</td><td>16#200</td><td>512</td></tr></table>			Encoding	Spacing Between Vectors (hex)	Spacing Between Vectors (decimal)	16#00	16#000	0	16#01	16#020	32	16#02	16#040	64	16#04	16#080	128	16#08	16#100	256	16#10	16#200	512
		Encoding			Spacing Between Vectors (hex)	Spacing Between Vectors (decimal)																			
		16#00			16#000	0																			
		16#01			16#020	32																			
		16#02			16#040	64																			
		16#04			16#080	128																			
		16#08			16#100	256																			
		16#10			16#200	512																			
All other values are reserved. The operation of the processor is <b>UNDEFINED</b> if a reserved value is written to this field.																									
0	4:0	Must be written as zero; returns zero on read.	0	0																					

### 5.2.8 SRSCtl Register (CP0 Register 12, Select 2)

The *SRSCtl* register controls the operation of GPR shadow sets in the processor. This register does not exist in implementations of the architecture prior to Release 2.

Figure 5.8 shows the format of the *SRSCtl* register; Table 5.10 describes the *SRSCtl* register fields.

**Figure 5.8 SRSCtl Register Format**

31	30	29	26	25	22	21	18	17	16	15	12	11	10	9	6	5	4	3	0
0 00	HSS		0 00 00	EICSS		0 00	ESS		0 00	PSS		0 00	CSS						

**Table 5.10 SRSCtl Register Field Descriptions**

Fields		Description	Read/Write	Reset State														
Name	Bits																	
0	31:30	Must be written as zeros; returns zero on read.	0	0														
HSS	29:26	<div>Highest Shadow Set. This field contains the highest shadow set number that is implemented by this processor. A value of zero in this field indicates that only the normal GPRs are implemented.</div> <div>Possible values of this field for the M14K processor are:</div> <table><thead><tr><th>Encoding</th><th>Meaning</th></tr></thead><tbody><tr><td>0</td><td>One shadow set (normal GPR set) is present.</td></tr><tr><td>1</td><td>Two shadow sets are present.</td></tr><tr><td>3</td><td>Four shadow sets are present.</td></tr><tr><td>7</td><td>Eight shadow sets are present</td></tr><tr><td>15</td><td>Sixteen shadow sets are present</td></tr><tr><td>2, 4-6, 8-14</td><td>Reserved</td></tr></tbody></table> <div>The value in this field also represents the highest value that can be written to the <i>ESS</i>, <i>EICSS</i>, <i>PSS</i>, and <i>CSS</i> fields of this register, or to any of the fields of the <i>SRSSMap</i> register. The operation of the processor is <b>UNDEFINED</b> if a value larger than the one in this field is written to any of these other fields.</div>	Encoding	Meaning	0	One shadow set (normal GPR set) is present.	1	Two shadow sets are present.	3	Four shadow sets are present.	7	Eight shadow sets are present	15	Sixteen shadow sets are present	2, 4-6, 8-14	Reserved	R	Preset
Encoding	Meaning																	
0	One shadow set (normal GPR set) is present.																	
1	Two shadow sets are present.																	
3	Four shadow sets are present.																	
7	Eight shadow sets are present																	
15	Sixteen shadow sets are present																	
2, 4-6, 8-14	Reserved																	
0	25:22	Must be written as zeros; returns zero on read.	0	0														
EICSS	21:18	EIC interrupt mode shadow set. If <i>Config3</i> <sub>VEIC</sub> is 1 (EIC interrupt mode is enabled), this field is loaded from the external interrupt controller for each interrupt request and is used in place of the <i>SRSSMap</i> register to select the current shadow set for the interrupt. See <a href="#">Section 4.3.1 “Interrupt Modes”</a> for a discussion of EIC interrupt mode. If <i>Config3</i> <sub>VEIC</sub> is 0, this field must be written as zero, and returns zero on read.	R	Undefined														
0	17:16	Must be written as zeros; returns zero on read.	0	0														
ESS	15:12	Exception Shadow Set. This field specifies the shadow set to use on entry to Kernel Mode caused by any exception other than a vectored interrupt. The operation of the processor is <b>UNDEFINED</b> if software writes a value into this field that is greater than the value in the <i>HSS</i> field.	R/W	0														
0	11:10	Must be written as zeros; returns zero on read.	0	0														

Table 5.10 SRSCtl Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
PSS	9:6	<p>Previous Shadow Set. If GPR shadow registers are implemented, and with the exclusions noted in the next paragraph, this field is copied from the <i>CSS</i> field when an exception or interrupt occurs. An ERET instruction copies this value back into the <i>CSS</i> field if <i>Status</i><sub>BEV</sub> = 0.</p> <p>This field is not updated on any exception which sets <i>Status</i><sub>ERL</sub> to 1 (i.e., Reset, Soft Reset, NMI, cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with <i>Status</i><sub>EXL</sub> = 1, or <i>Status</i><sub>BEV</sub> = 1. This field is not updated on an exception that occurs while <i>Status</i><sub>ERL</sub> = 1.</p> <p>The operation of the processor is <b>UNDEFINED</b> if software writes a value into this field that is greater than the value in the <i>HSS</i> field.</p>	R/W	0
0	5:4	Must be written as zeros; returns zero on read.	0	0
CSS	3:0	<p>Current Shadow Set. If GPR shadow registers are implemented, this field is the number of the current GPR set. With the exclusions noted in the next paragraph, this field is updated with a new value on any interrupt or exception, and restored from the <i>PSS</i> field on an ERET. Table 5.11 describes the various sources from which the <i>CSS</i> field is updated on an exception or interrupt.</p> <p>This field is not updated on any exception which sets <i>Status</i><sub>ERL</sub> to 1 (i.e., Reset, Soft Reset, NMI, cache error), an entry into EJTAG Debug mode, or any exception or interrupt that occurs with <i>Status</i><sub>EXL</sub> = 1, or <i>Status</i><sub>BEV</sub> = 1. Neither is it updated on an ERET with <i>Status</i><sub>ERL</sub> = 1 or <i>Status</i><sub>BEV</sub> = 1. This field is not updated on an exception that occurs while <i>Status</i><sub>ERL</sub> = 1.</p> <p>The value of <i>CSS</i> can be changed directly by software only by writing the <i>PSS</i> field and executing an ERET instruction.</p>	R	0

### 5.2.9 SRSSMap Register (CP0 Register 12, Select 3)

**Table 5.11 Sources for new  $SRSCtl_{CSS}$  on an Exception or Interrupt**

Exception Type	Condition	$SRSCtl_{CSS}$ Source	Comment
Exception	All	$SRSCtl_{ESS}$	
Non-Vectored Interrupt	$Cause_{IV} = 0$	$SRSCtl_{ESS}$	Treat as exception
Vectored Interrupt	$Cause_{IV} = 1$ and $Config3_{VEIC} = 0$ and $Config3_{VInt} = 1$	$SRSSMap_{VECTNUM}$	Source is internal map register. (for VECTNUM see Table 4.3)
Vectored EIC Interrupt	$Cause_{IV} = 1$ and $Config3_{VEIC} = 1$	$SRSCtl_{EICSS}$	Source is external interrupt controller.

The  $SRSSMap$  register contains 8, 4-bit fields that provide the mapping from a vector number to the shadow set number to use when servicing such an interrupt. The values from this register are not used for a non-interrupt exception, or a non-vectored interrupt ( $Cause_{IV} = 0$  or  $IntCtl_{VS} = 0$ ). In such cases, the shadow set number comes from  $SRSCtl_{ESS}$ .

If  $SRSCtl_{HSS}$  is zero, the results of a software read or write of this register are **UNPREDICTABLE**.

The operation of the processor is **UNDEFINED** if a value is written to any field in this register that is greater than the value of  $SRSCtl_{HSS}$ .

The  $SRSSMap$  register contains the shadow register set numbers for vector numbers 7:0. The same shadow set number can be established for multiple interrupt vectors, creating a many-to-one mapping from a vector to a single shadow register set number.

Figure 5.9 shows the format of the  $SRSSMap$  register; Table 5.12 describes the  $SRSSMap$  register fields.

**Figure 5.9 SRSSMap Register Format**

31	28	27	24	23	20	19	16	15	12	11	8	7	4	3	0
SSV7	SSV6	SSV5	SSV4	SSV3	SSV2	SSV1	SSV0								

**Table 5.12 SRSSMap Register Field Descriptions**

Fields		Description	Read/Write	Reset State
Name	Bits			
SSV7	31:28	Shadow register set number for Vector Number 7	R/W	0
SSV6	27:24	Shadow register set number for Vector Number 6	R/W	0
SSV5	23:20	Shadow register set number for Vector Number 5	R/W	0
SSV4	19:16	Shadow register set number for Vector Number 4	R/W	0
SSV3	15:12	Shadow register set number for Vector Number 3	R/W	0
SSV2	11:8	Shadow register set number for Vector Number 2	R/W	0
SSV1	7:4	Shadow register set number for Vector Number 1	R/W	0
SSV0	3:0	Shadow register set number for Vector Number 0	R/W	0

### 5.2.10 View\_IPL Register (CP0 Register 12, Select 4)

Figure 5-10 View\_IPL Register Format

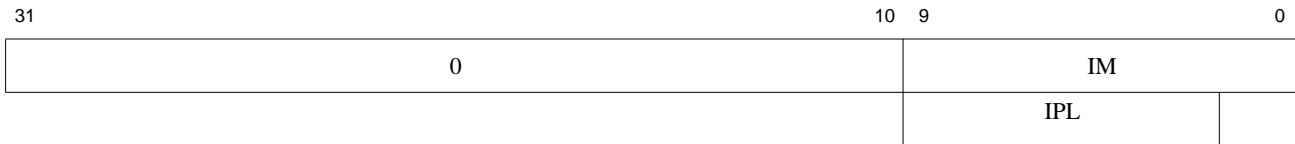


Table 5.13 View\_IPL Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
IM	9:0	Interrupt Mask. If EIC interrupt mode is not enabled, controls which interrupts are enabled.	R/W	Undefined for IM7:IM2 0 for IM9:IM8	Required
IPL	9:2	Interrupt Priority Level. If EIC interrupt mode is enabled, this field is the encoded value of the current <i>IPL</i> .	R/W	Undefined	Required
0	31:10,1:0	Must be written as zero; returns zero on read.	0	0	Reserved

This register gives read and write access to the *IM* or *IPL* field that is also available in the *Status* Register. The use of this register allows the Interrupt Mask or the Priority Level to be read/written without extracting/inserting that bit field from/to the *Status* Register.

The *IPL* field might be located in non-contiguous bits within the *Status* Register. All of the *IPL* bits are presented as a contiguous field within this register.

### 5.2.11 SRSSMap2 Register (CP0 Register 12, Select 5)

The *SRSSMap2* register contains 2 4-bit fields that provide the mapping from an vector number to the shadow set number to use when servicing such an interrupt. The values from this register are not used for a non-interrupt exception, or a non-vectored interrupt ( $Cause_{IV} = 0$  or  $IntCtl_{VS} = 0$ ). In such cases, the shadow set number comes from  $SRSCtl_{ESS}$ .

If  $SRSCtl_{HSS}$  is zero, the results of a software read or write of this register are **UNPREDICTABLE**.

The operation of the processor is **UNDEFINED** if a value is written to any field in this register that is greater than the value of  $SRSCtl_{HSS}$ .

The *SRSSMap2* register contains the shadow register set numbers for vector numbers 9:8. The same shadow set number can be established for multiple interrupt vectors, creating a many-to-one mapping from a vector to a single shadow register set number.



Figure 5-11 shows the format of the *SRSMap2* register; Table 5.14 describes the *SRSMap2* register fields.

**Figure 5-11 SRSMap Register Format**



**Table 5.14 SRSMap Register Field Descriptions**

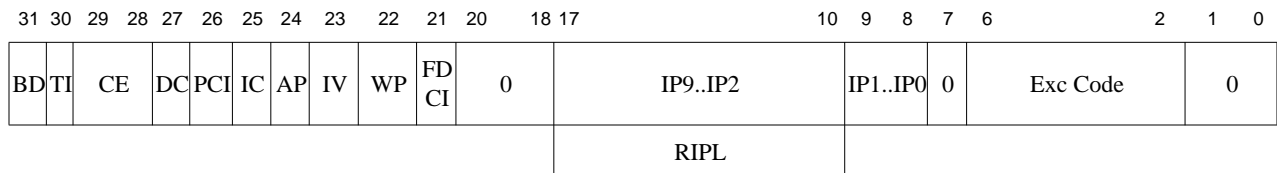
Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
0	31:8	Must be written as zero; returns zero on read.	R	0	RESERVED
SSV9	7:4	Shadow register set number for Vector Number 9	R/W	0	Required
SSV8	3:0	Shadow register set number for Vector Number 8	R/W	0	Required

### 5.2.12 Cause Register (CP0 Register 13, Select 0)

The *Cause* register primarily describes the cause of the most recent exception. In addition, fields also control software interrupt requests and the vector through which interrupts are dispatched. With the exception of the *IP1..0*, *DC*, *IV*, and *WP* fields, all fields in the *Cause* register are read-only. Release 2 of the Architecture added optional support for an External Interrupt Controller (EIC) interrupt mode, in which *IP7..2* are interpreted as the Requested Interrupt Priority Level (RIPL).

Figure 5.12 shows the format of the *Cause* register; Table 5.15 describes the *Cause* register fields.

**Figure 5.12 Cause Register Format**



**Table 5.15 Cause Register Field Descriptions**

Fields		Description	Read/Write	Reset State						
Name	Bits									
BD	31	Indicates whether the last exception taken occurred in a branch delay slot:	R	Undefined						
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Not in delay slot</td></tr><tr><td>1</td><td>In delay slot</td></tr></table>			Encoding	Meaning	0	Not in delay slot	1	In delay slot
		Encoding			Meaning					
		0			Not in delay slot					
		1			In delay slot					
	The processor updates <i>BD</i> only if <i>Status</i> <sub>EXL</sub> was zero when the exception occurred.									

Table 5.15 Cause Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State						
Name	Bits									
TI	30	<p>Timer Interrupt. This bit denotes whether a timer interrupt is pending (analogous to the <i>IP</i> bits for other interrupt types):</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No timer interrupt is pending</td></tr><tr><td>1</td><td>Timer interrupt is pending</td></tr></table> <p>The state of the <i>TI</i> bit is available on the external core interface as the <i>SL_TimerInt</i> signal</p>	Encoding	Meaning	0	No timer interrupt is pending	1	Timer interrupt is pending	R	Undefined
Encoding	Meaning									
0	No timer interrupt is pending									
1	Timer interrupt is pending									
CE	29:28	<p>Coprocessor unit number referenced when a Coprocessor Unusable exception is taken. This field is loaded by hardware on every exception, but is <b>UNPREDICTABLE</b> for all exceptions except for Coprocessor Unusable.</p>	R	Undefined						
DC	27	<p>Disable Count register. In some power-sensitive applications, the <i>Count</i> register is not used and is the source of meaningful power dissipation. This bit allows the <i>Count</i> register to be stopped in such situations.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Enable counting of <i>Count</i> register</td></tr><tr><td>1</td><td>Disable counting of <i>Count</i> register</td></tr></table>	Encoding	Meaning	0	Enable counting of <i>Count</i> register	1	Disable counting of <i>Count</i> register	R/W	0
Encoding	Meaning									
0	Enable counting of <i>Count</i> register									
1	Disable counting of <i>Count</i> register									
PCI	26	<p>Performance Counter Interrupt. In an implementation of Release 2 of the Architecture, this bit denotes whether a performance counter interrupt is pending (analogous to the IP bits for other interrupt types):</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No timer interrupt is pending</td></tr><tr><td>1</td><td>Timer interrupt is pending</td></tr></table> <p>The state of the <i>PCI</i> bit is available on the external M14K interface as the <i>SL_PCInt</i> signal.</p>	Encoding	Meaning	0	No timer interrupt is pending	1	Timer interrupt is pending	R	0
Encoding	Meaning									
0	No timer interrupt is pending									
1	Timer interrupt is pending									
IC	25	<p>Indicates if Interrupt Chaining occurred on the last IRET instruction.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupt Chaining did not happen on last IRET</td></tr><tr><td>1</td><td>Interrupt Chaining occurred during last IRET</td></tr></table>	Encoding	Meaning	0	Interrupt Chaining did not happen on last IRET	1	Interrupt Chaining occurred during last IRET	R	Undefined
Encoding	Meaning									
0	Interrupt Chaining did not happen on last IRET									
1	Interrupt Chaining occurred during last IRET									

Table 5.15 Cause Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State						
Name	Bits									
AP	24	Indicates whether an exception occurred during Interrupt Auto-Prologue.	R	Undefined						
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Exception did not occur during Auto-Prologue operation.</td></tr><tr><td>1</td><td>Exception occurred during Auto-Prologue operation.</td></tr></table>			Encoding	Meaning	0	Exception did not occur during Auto-Prologue operation.	1	Exception occurred during Auto-Prologue operation.
		Encoding			Meaning					
		0			Exception did not occur during Auto-Prologue operation.					
1	Exception occurred during Auto-Prologue operation.									
IV	23	Indicates whether an interrupt exception uses the general exception vector or a special interrupt vector:	R/W	Undefined						
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Use the general exception vector (16#180)</td></tr><tr><td>1</td><td>Use the special interrupt vector (16#200)</td></tr></table>			Encoding	Meaning	0	Use the general exception vector (16#180)	1	Use the special interrupt vector (16#200)
		Encoding			Meaning					
		0			Use the general exception vector (16#180)					
1	Use the special interrupt vector (16#200)									
	In implementations of Release 2 of the architecture, if the <i>Cause</i> <sub>IV</sub> is 1 and <i>Status</i> <sub>BEV</sub> is 0, the special interrupt vector represents the base of the vectored interrupt table.									
WP	22	Indicates that a watch exception was deferred because <i>Status</i> <sub>EXL</sub> or <i>Status</i> <sub>ERL</sub> were a one at the time the watch exception was detected. This bit both indicates that the watch exception was deferred, and causes the exception to be initiated once <i>Status</i> <sub>EXL</sub> and <i>Status</i> <sub>ERL</sub> are both zero. As such, software must clear this bit as part of the watch exception handler to prevent a watch exception loop. Software should not write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is <b>UNPREDICTABLE</b> whether hardware ignores the write, accepts the write with no side effects, or accepts the write and initiates a watch exception once <i>Status</i> <sub>EXL</sub> and <i>Status</i> <sub>ERL</sub> are both zero. Because watch registers are not implemented on the M14K core, this bit is ignored on writes and reads as zero.	R	0						

Table 5.15 Cause Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State																											
Name	Bits																														
FDCI	21	<p>Fast Debug Channel Interrupt. This bit denotes whether a FDC Interrupt is pending (analogous to the IP bits for other interrupt types):</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No Fast Debug Channel interrupt is pending</td></tr><tr><td>1</td><td>Fast Debug Channel interrupt is pending</td></tr></table>	Encoding	Meaning	0	No Fast Debug Channel interrupt is pending	1	Fast Debug Channel interrupt is pending	R	Undefined																					
Encoding	Meaning																														
0	No Fast Debug Channel interrupt is pending																														
1	Fast Debug Channel interrupt is pending																														
IP9..IP2	17:10	<p>Indicates an interrupt is pending:</p> <table><tr><th>Bit</th><th>Name</th><th>Meaning</th></tr><tr><td>17</td><td>IP9</td><td>Hardware Interrupt 7</td></tr><tr><td>16</td><td>IP8</td><td>Hardware Interrupt 6</td></tr><tr><td>15</td><td>IP7</td><td>Hardware interrupt 5</td></tr><tr><td>14</td><td>IP6</td><td>Hardware interrupt 4</td></tr><tr><td>13</td><td>IP5</td><td>Hardware interrupt 3</td></tr><tr><td>12</td><td>IP4</td><td>Hardware interrupt 2</td></tr><tr><td>11</td><td>IP3</td><td>Hardware interrupt 1</td></tr><tr><td>10</td><td>IP2</td><td>Hardware interrupt 0</td></tr></table> <p>In implementations of Release 1 of the Architecture, timer and performance counter interrupts are combined in an implementation-dependent way with hardware interrupt 5.</p> <p>In implementations of Release 2 of the Architecture in which EIC interrupt mode is not enabled (<i>Config3</i><sub>VEIC</sub> = 0), timer and performance counter interrupts are combined in an implementation-dependent way with any hardware interrupt. If EIC interrupt mode is enabled (<i>Config3</i><sub>VEIC</sub> = 1), these bits take on a different meaning and are interpreted as the <i>RIPL</i> field, described below.</p>	Bit	Name	Meaning	17	IP9	Hardware Interrupt 7	16	IP8	Hardware Interrupt 6	15	IP7	Hardware interrupt 5	14	IP6	Hardware interrupt 4	13	IP5	Hardware interrupt 3	12	IP4	Hardware interrupt 2	11	IP3	Hardware interrupt 1	10	IP2	Hardware interrupt 0	R	Undefined for IP7:IP2  0 for IP9:IP8
Bit	Name	Meaning																													
17	IP9	Hardware Interrupt 7																													
16	IP8	Hardware Interrupt 6																													
15	IP7	Hardware interrupt 5																													
14	IP6	Hardware interrupt 4																													
13	IP5	Hardware interrupt 3																													
12	IP4	Hardware interrupt 2																													
11	IP3	Hardware interrupt 1																													
10	IP2	Hardware interrupt 0																													
RIPL	17:10	<p>Requested Interrupt Priority Level.</p> <p>In implementations of Release 2 of the Architecture in which EIC interrupt mode is enabled (<i>Config3</i><sub>VEIC</sub> = 1), this field is the encoded (0..255) value of the requested interrupt. A value of zero indicates that no interrupt is requested.</p> <p>If EIC interrupt mode is not enabled (<i>Config3</i><sub>VEIC</sub> = 0), these bits take on a different meaning and are interpreted as the <i>IP7..IP2</i> bits, described above.</p>	R	Undefined for bits 15:10  0 for bits 17:16																											

Table 5.15 Cause Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State									
Name	Bits												
IP1..IP0	9:8	Controls the request for software interrupts:	R/W	Undefined									
		<table><tr><th>Bit</th><th>Name</th><th>Meaning</th></tr><tr><td>9</td><td>IP1</td><td>Request software interrupt 1</td></tr><tr><td>8</td><td>IP0</td><td>Request software interrupt 0</td></tr></table>			Bit	Name	Meaning	9	IP1	Request software interrupt 1	8	IP0	Request software interrupt 0
		Bit			Name	Meaning							
		9			IP1	Request software interrupt 1							
		8			IP0	Request software interrupt 0							
These bits are exported to an external interrupt controller for prioritization in EIC interrupt mode with other interrupt sources. The state of these bits is available on the external core interface as the <i>SI_SWInt[1:0]</i> bus.													
ExcCode	6:2	Exception code - see <a href="#">Table 5.16</a>	R	Undefined									
0	20:18, 7, 1:0	Must be written as zero; returns zero on read.	0	0									

Table 5.16 Cause Register ExcCode Field

Exception Code Value		Mnemonic	Description
Decimal	Hexadecimal		
0	16#00	Int	Interrupt
1-3	16#01-16#03	-	Reserved
4	16#04	AdEL	Address error exception (load or instruction fetch)
5	16#05	AdES	Address error exception (store)
6	16#06	IBE	Bus error exception (instruction fetch)
7	16#07	DBE	Bus error exception (data reference: load or store)
8	16#08	Sys	Syscall exception
9	16#09	Bp	Breakpoint exception
10	16#0a	RI	Reserved instruction exception
11	16#0b	CpU	Coprocessor Unusable exception
12	16#0c	Ov	Arithmetic Overflow exception
13	16#0d	Tr	Trap exception
14-15	16#0e-16#0f	-	Reserved
16	16#10	IS1	Implementation-Specific Exception 1 (COP2)
17	16#11	CEU	CorExtend Unusable
18	16#12	C2E	Coprocessor 2 exceptions
19-28	16#13-16#1c	-	Reserved
29	16#1d	MPU	MPU Exception
30	16#1e	Parity Error	Parity error. In normal mode, a parity error exception has a dedicated vector and the Cause register is not updated. If a parity error occurs while in Debug Mode, this code is written to the <i>DebugDExcCode</i> field to indicate that re-entry to Debug Mode was caused by a parity error

Table 5.16 Cause Register ExcCode Field (Continued)

Exception Code Value		Mnemonic	Description
Decimal	Hexadecimal		
31	16#1f	-	Reserved

### 5.2.13 View\_RIPL Register (CP0 Register 13, Select 4)

Figure 5-13 View\_RIPL Register Format

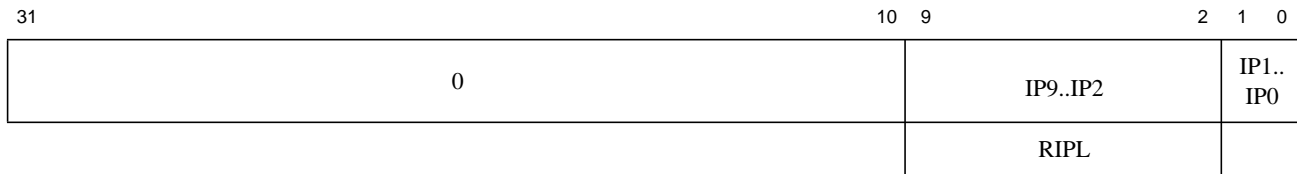


Table 5.17 View\_RIPL Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
IP1..IP0	1:0	SW Interrupt Pending. If EIC interrupt mode is not enabled, controls which SW interrupts are pending.	R/W	Undefined	Required
IP9..IP2	9:2	HW Interrupt Pending. If EIC interrupt mode is not enabled, indicates which HW interrupts are pending.	R	Undefined for IP7:IP2  0 for IP9:IP8	Required
RIPL	9:2	Interrupt Priority Level. If EIC interrupt mode is enabled, this field indicates the Requested Priority Level of the pending interrupt.	R	Undefined	Required
0	31:10,1:0	Must be written as zero; returns zero on read.	0	0	Reserved

This register gives read access to the *IP* or *RIPL* field that is also available in the *Cause* Register. The use of this register allows the Interrupt Pending or the Requested Priority Level to be read without extracting that bit field from the *Cause* Register.

### 5.2.14 NestedExc (CP0 Register 13, Select 5)

**Compliance Level:** *Optional.*

The *Nested Exception* (*NestedExc*) register is a read-only register containing the values of *Status<sub>EXL</sub>* and *Status<sub>ERL</sub>* prior to acceptance of the current exception.

This register is part of the Nested Fault feature. The existence of the register can be determined by reading the *Config5<sub>NFExists</sub>* bit.

Figure 5-14 shows the format of the *NestedExc* register; Table 5.18 describes the *NestedExc* register fields.

**Figure 5-14 NestedExc Register Format**



**Table 5.18 NestedExc Register Field Descriptions**

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
0	31..3	Reserved, read as 0.	R0	0	Required
ERL	2	Value of <i>Status<sub>ERL</sub></i> prior to acceptance of current exception.  Updated by all exceptions that would set either <i>Status<sub>EXL</sub></i> or <i>Status<sub>ERL</sub></i> . Not updated by Debug exceptions.	R	Undefined	Required
EXL	1	Value of <i>Status<sub>EXL</sub></i> prior to acceptance of current exception.  Updated by exceptions which would update EPC if <i>Status<sub>EXL</sub></i> is not set (MCheck, Interrupt, Address Error, all TLB exceptions, Bus Error, CopUnusable, Reserved Instruction, Overflow, Trap, Syscall, FPU, etc.) . For these exception types, this register field is updated regardless of the value of <i>Status<sub>EXL</sub></i> .  Not updated by exception types which update <i>ErrorEPC</i> - (Reset, Soft Reset, NMI, Cache Error). Not updated by Debug exceptions.	R	Undefined	Required
0	0	Reserved, read as 0.	R0	0	Required

### 5.2.15 Exception Program Counter (CP0 Register 14, Select 0)

The *Exception Program Counter (EPC)* is a read/write register that contains the address at which processing resumes after an exception has been serviced. All bits of the *EPC* register are significant and must be writable.

For synchronous (precise) exceptions, the *EPC* contains one of the following:

- The virtual address of the instruction that was the direct cause of the exception

- The virtual address of the immediately preceding branch or jump instruction, when the exception causing instruction is in a branch delay slot and the *Branch Delay* bit in the *Cause* register is set.

On new exceptions, the processor does not write to the *EPC* register when the *EXL* bit in the *Status* register is set, however, the register can still be written via the *MTC0* instruction.

In processors that implement microMIPS, a read of the *EPC* register (via *MFC0*) returns the following value in the destination GPR:

$$\text{GPR}[\text{rt}] \leftarrow \text{ExceptionPC}_{31..1} \parallel \text{ISAMode}_0$$

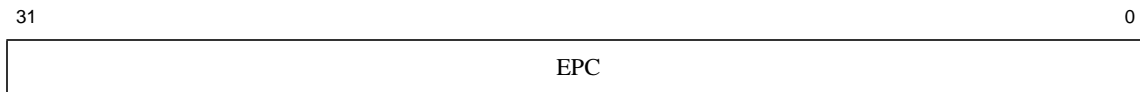
That is, the upper 31 bits of the exception PC are combined with the lower bit of the *ISAMode* field and written to the GPR.

Similarly, a write to the *EPC* register (via *MTC0*) takes the value from the GPR and distributes that value to the exception PC and the *ISAMode* field, as follows

$$\begin{aligned} \text{ExceptionPC} &\leftarrow \text{GPR}[\text{rt}]_{31..1} \parallel 0 \\ \text{ISAMode} &\leftarrow 2\#0 \parallel \text{GPR}[\text{rt}]_0 \end{aligned}$$

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the exception PC, and the lower bit of the exception PC is cleared. The upper bit of the *ISAMode* field is cleared and the lower bit is loaded from the lower bit of the GPR.

**Figure 5.15 EPC Register Format**



**Table 5.19 EPC Register Field Description**

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
EPC	31:0	Exception Program Counter.	R/W	Undefined

### 5.2.16 NestedEPC (CP0 Register 14, Select 2)

**Compliance Level:** *Optional*.

The *Nested Exception Program Counter* (*NestedEPC*) is a read/write register with the same behavior as the *EPC* register except that:

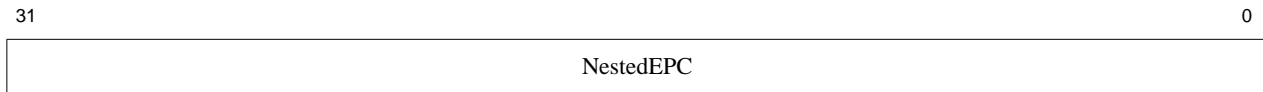
- The *NestedEPC* register ignores the value of *StatusEXL* and is therefore updated on the occurrence of any exception, including nested exceptions.
- The *NestedEPC* register is not used by the *ERET/DERET/IRET* instructions. To return to the address stored in *NestedEPC*, software must copy the value of the *NestedEPC* register to the *EPC* register.

This register is part of the Nested Fault feature. The existence of the register can be determined by reading the *Config5<sub>NFExists</sub>* bit.



Figure 5-14 shows the format of the *NestedEPC* register; Table 5.18 describes the *NestedEPC* register fields.

**Figure 5-16 NestedEPC Register Format**



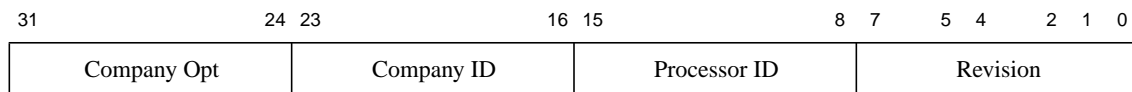
**Table 5.20 NestedEPC Register Field Descriptions**

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
NestedEPC	31..0	<p>Nested Exception Program Counter</p> <p>Updated by exceptions which would update EPC if <i>StatusEXL</i> is not set (MCheck, Interrupt, Address Error, all TLB exceptions, Bus Error, CopUnusable, Reserved Instruction, Overflow, Trap, Syscall, FPU, etc.) . For these exception types, this register field is updated regardless of the value of <i>StatusEXL</i>.</p> <p>Not updated by exception types which update <i>ErrorEPC</i> - (Reset, Soft Reset, NMI, Cache Error). Not updated by Debug exceptions.</p>	R/W	Undefined	Required

### 5.2.17 Processor Identification (CP0 Register 15, Select 0)

The *Processor Identification (PRId)* register is a 32 bit read-only register that contains information identifying the manufacturer, manufacturer options, processor identification, and revision level of the processor.

**Figure 5.17 PRId Register Format**



**Table 5.21 PRId Register Field Descriptions**

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Company Opt	31:24	<p>Company Option. Whatever is specified by the SoC builder who synthesizes the M14K - refer to your SoC manual. It should be preset at config GUI with a number between 0x00 and 0x7F— higher values (0x80-0xFF) are reserved by MIPS Technologies.</p>	R	Preset

Table 5.21 PRId Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
Company ID	23:16	Identifies the company that designed or manufactured the processor. In the M14K this field contains a value of 1 to indicate MIPS Technologies, Inc.	R	1
Processor ID	15:8	Identifies the type of processor. This field allows software to distinguish between the various types of MIPS Technologies processors.	R	0x9B
Revision	7:0	Specifies the revision number of the processor. This field allows software to distinguish between one revision and another of the same processor type. This field is broken up into the following three subfields	R	Preset
Major Revision	7:5	This number is increased on major revisions of the processor core	R	Preset
Minor Revision	4:2	This number is increased on each incremental revision of the processor and reset on each new major revision	R	Preset
Patch Level	1:0	If a patch is made to modify an older revision of the processor, this field will be incremented	R	Preset

### 5.2.18 EBase Register (CP0 Register 15, Select 1)

The *EBase* register is a read/write register containing the base address of the exception vectors used when *Status<sub>BEV</sub>* equals 0, and a read-only CPU number value that may be used by software to distinguish different processors in a multi-processor system.

The *EBase* register provides the ability for software to identify the specific processor within a multi-processor system, and allows the exception vectors for each processor to be different, especially in systems composed of heterogeneous processors. Bits 31:12 of the *EBase* register are concatenated with zeros to form the base of the exception vectors when *Status<sub>BEV</sub>* is 0. The exception vector base address comes from the fixed defaults (see [Section 4.5 “Exception Vector Locations”](#)) when *Status<sub>BEV</sub>* is 1, or for any EJTAG Debug exception. The reset state of bits 31:12 of the *EBase* register initialize the exception base register to 16#8000.0000, providing backward compatibility with Release 1 implementations.

Bits 31:30 of the *EBase* Register are fixed with the value 2#10 to force the exception base address to be in the kseg0 or kseg1 unmapped virtual address segments.

If the value of the exception base register is to be changed, this must be done with *Status<sub>BEV</sub>* equal 1. The operation of the processor is **UNDEFINED** if the Exception Base field is written with a different value when *Status<sub>BEV</sub>* is 0.

Combining bits 31:20 with the *Exception Base* field allows the base address of the exception vectors to be placed at any 4KByte page boundary. If vectored interrupts are used, a vector offset greater than 4KBytes can be generated. In this case, bit 12 of the *Exception Base* field must be zero. The operation of the processor is **UNDEFINED** if software writes bit 12 of the *Exception Base* field with a 1 and enables the use of a vectored interrupt whose offset is greater than 4KBytes from the exception base address.

Figure 5.18 shows the format of the *EBase* Register; Table 5.22 describes the *EBase* register fields.

### Figure 5.18 EBase Register Format

31	30	29					12	11	10	9					0
1	0	Exception Base						0	0	CPUNum					

### Table 5.22 EBase Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
1	31	This bit is ignored on writes and returns one on reads.	R	1
0	30	This bit is ignored on writes and returns zero on reads.	R	0
Exception Base	29:12	In conjunction with bits 31:30, this field specifies the base address of the exception vectors when <i>Status</i> <sub>BEV</sub> is zero.	R/W	0
0	11:10	Must be written as zero; returns zero on reads.	0	0
CPUNum	9:0	This field specifies the number of the CPU in a multi-processor system and can be used by software to distinguish a particular processor from the others. The value in this field is set by the <i>SL_CPUNum[9:0]</i> static input pins to the core. In a single processor system, this value should be set to zero.	R	Externally Set

### 5.2.19 CDMMBase Register (CP0 Register 15, Select 2)

The 36-bit physical base address for the Common Device Memory Map facility is defined by this register. This register only exists if *Config3<sub>CDMM</sub>* is set to one.

Figure 5.19 shows the format of the *CDMMBase* register, and Table 5.23 describes the register fields.

### Figure 5.19 CDMMBase Register Format

31																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																													
----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

### Table 5.23 CDMMBase Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
CDMM_UP PER_ADDR	31:11	<p>Bits 35:15 of the base physical address of the memory mapped registers.</p> <p>The number of implemented physical address bits is implementation-specific. For the unimplemented address bits, writes are ignored and reads return zero.</p>	R/W	Undefined

Table 5.23 CDMMBase Register Field Descriptions

Fields		Description	Read/Write	Reset State												
Name	Bits															
EN	10	<p>Enables the CDMM region.</p> <p>If this bit is cleared, memory requests to this address region access regular system memory. If this bit is set, memory requests to this region access the CDMM logic</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>CDMM Region is disabled.</td></tr><tr><td>1</td><td>CDMM Region is enabled.</td></tr></table>	Encoding	Meaning	0	CDMM Region is disabled.	1	CDMM Region is enabled.	R/W	0						
Encoding	Meaning															
0	CDMM Region is disabled.															
1	CDMM Region is enabled.															
CI	9	If set to 1, this indicates that the first 64-byte Device Register Block of the CDMM is reserved for additional registers that manage CDMM region behavior and are not IO device registers.	R	Preset												
CDMMSize	8:0	<p>This field represents the number of 64-byte Device Register Blocks instantiated in the core.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>1 DRB</td></tr><tr><td>1</td><td>2 DRBs</td></tr><tr><td>2</td><td>3 DRBs</td></tr><tr><td>...</td><td>...</td></tr><tr><td>511</td><td>512 DRBs</td></tr></table>	Encoding	Meaning	0	1 DRB	1	2 DRBs	2	3 DRBs	...	...	511	512 DRBs	R	Preset
Encoding	Meaning															
0	1 DRB															
1	2 DRBs															
2	3 DRBs															
...	...															
511	512 DRBs															

### 5.2.20 Config Register (CP0 Register 16, Select 0)

The *Config* register specifies various configuration and capabilities information. Most of the fields in the *Config* register are initialized by hardware during the Reset exception process, or are constant.

Figure 5.20 Config Register Format — Select 0

31	30	28	27	25	24	23	22	21	20	19	17	16	15	14	13	12	10	9	7	6	3	2	0
M	K23	KU	0	UDI	SB	MDU	0	DS	BE	AT	AR	MT	0	K0									

Figure 5.21 Config Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
M	31	This bit is hardwired to '1' to indicate the presence of the Config1 register.	R	1

Figure 5.21 Config Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
K23	30:28	This field controls the cacheability of the kseg2 and kseg3 address segments in FM implementations. Refer to <a href="#">Table 5.24</a> for the field encoding.	FM: R/W	FM: 010
KU	27:25	This field controls the cacheability of the kuseg and useg address segments in FM implementations. Refer to <a href="#">Table 5.24</a> for the field encoding.	FM: R/W	FM: 010
0	24:23	Must be written as 0. Returns zero on reads.	0	0
UDI	22	This bit indicates that CorExtend User Defined Instructions have been implemented. 0 = No User Defined Instructions are implemented 1 = User Defined Instructions are implemented	R	Preset
SB	21	Indicates whether SimpleBE bus mode is enabled. Set via <a href="#">SL_SimpleBE[0]</a> input pin: 0 = No reserved byte enables on SRAM interface 1 = Only simple byte enables allowed on SRAM interface	R	Externally Set
MDU	20	This bit indicates the type of Multiply/Divide Unit present. 0 = Fast, high-performance MDU 1 = Iterative, area-efficient MDU	R	Preset
0	19:17	Must be written as 0. Returns zero on reads.	0	0
DS	16	Dual SRAM interface. 0: Unified instruction/data SRAM interface 1: Dual instruction/data SRAM interfaces	R	Preset
BE	15	Indicates the endian mode in which the processor is running. Set via <a href="#">SL_Endian</a> input pin. 0: Little endian 1: Big endian	R	Externally Set
AT	14:13	Architecture type implemented by the processor. This field is always 00 to indicate the MIPS32 architecture.	R	00
AR	12:10	Architecture revision level. This field is always 001 to indicate MIPS32 Release 2. 0: Release 1 1: Release 2 2-7: Reserved	R	001
MT	9:7	MMU Type: 3: Fixed Mapping 0-2, 4-7: Reserved	R	3
0	6:3	Must be written as zeros; returns zeros on reads.	0	0
K0	2:0	Kseg0 coherency algorithm. Refer to <a href="#">Table 5.24</a> for the field encoding.	R/W	010

**Table 5.24 Cache Coherency Attributes**

C(2:0) Value	Cache Coherency Attribute
2	Uncached.
3	Cached (Core treats as uncached, but passes attribute to the system for use with any external caching mechanisms)

### 5.2.21 Config1 Register (CP0 Register 16, Select 1)

The *Config1* register is an adjunct to the *Config* register and encodes additional information about capabilities present on the core. All fields in the *Config1* register are read-only.

**Figure 5.22 Config1 Register Format — Select 1**

31	30	25	24	22	21	19	18	16	15	13	12	10	9	7	6	5	4	3	2	1	0
M	MMU Size	IS	IL	IA	DS	DL	DA	C2	MD	PC	WR	CA	EP	FP							

**Table 5.25 Config1 Register Field Descriptions — Select 1**

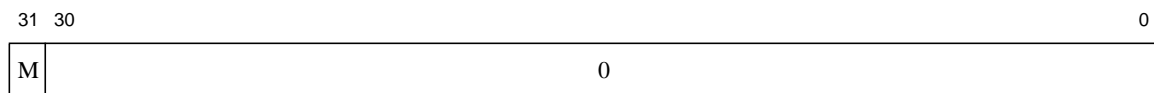
Fields		Description	Read/Write	Reset State
Name	Bit(s)			
M	31	This bit is hardwired to '1' to indicate the presence of the <i>Config2</i> register.	R	1
MMU Size	30:25	This field contains the number of entries in the TLB minus one.	R	0
IS	24:22	This field contains the number of instruction cache sets per way. Because the M14K core does not include caches, this field is always read as 0.	R	0
IL	21:19	This field contains the instruction cache line size. Because the M14K core does not include caches, this field is always read as 0.	R	0
IA	18:16	This field contains the level of instruction cache associativity. Because the M14K core does not include caches, this field is always read as 0.	R	0
DS	15:13	This field contains the number of data cache sets per way. Because the M14K core does not include caches, this field is always read as 0.	R	0
DL	12:10	This field contains the data cache line size. Because the M14K core does not include caches, this field is always read as 0.	R	0
DA	9:7	This field contains the type of set associativity for the data cache. Because the M14K core does not include caches, this field is always read as 0.	R	0

**Table 5.25 Config1 Register Field Descriptions — Select 1 (Continued)**

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
C2	6	Coprocessor 2 present. 0: No coprocessor is attached to the COP2 interface 1: A coprocessor is attached to the COP2 interface If the Cop2 interface logic is not implemented, this bit will read 0.	R	Preset
MD	5	MDMX implemented. This bit always reads as 0 because MDMX is not supported.	R	0
PC	4	Performance Counter registers implemented.	R	Preset
WR	3	Watch registers implemented. 0: No Watch registers are present 1: One or more Watch registers are present This bit is always read as 0, because the M14K core does not contain Watch registers.	R	0
CA	2	Code compression (MIPS16e) implemented. 0: No MIPS16e present 1: MIPS16e is implemented	R	0
EP	1	EJTAG present: This bit is always set to indicate that the core implements EJTAG.	R	1
FP	0	FPU implemented. This bit is always zero, because the core does not contain a floating point unit.	R	0

**5.2.22 Config2 Register (CP0 Register 16, Select 2)**

The *Config2* register is an adjunct to the *Config* register and is reserved to encode additional capabilities information. *Config2* is allocated for showing the configuration of level 2/3 caches. These fields are reset to 0 because L2/L3 caches are not supported by the M14K core. All fields in the *Config2* register are read-only.

**Figure 5.23 Config2 Register Format — Select 2****Table 5.26 Config2 Register Field Descriptions — Select 1**

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
M	31	This bit is hardwired to '1' to indicate the presence of the <i>Config3</i> register.	R	1
0	30:0	These bits are reserved.	R	0

### 5.2.23 Config3 Register (CP0 Register 16, Select 3)

The *Config3* register encodes additional capabilities. All fields in the *Config3* register are read-only.

Figure 5-24 shows the format of the *Config3* register; Table 5.27 describes the *Config3* register fields.

**Figure 5-24 Config3 Register Format**

31	30	23	22	21	20	18	17	16	15	14	13	12	11	9	8	7	6	5	4	3	2	1	0
M	000000000	IPLW	MMAR	MCU	ISA On Exc	ISA	ULRI	RXI	0	ITL	LPA	V E I C	V I n t	SP	CD M M	0	SM	TL					

**Table 5.27 Config3 Register Field Descriptions**

Fields		Description	Read/Write	Reset State								
Name	Bits											
M	31	This bit is reserved to indicate that a <i>Config4</i> register is present.	R	1								
0	30:23,11:9,2	Must be written as zeros; returns zeros on read	0	0								
IPLW	22:21	Width of the <i>Status<sub>IPL</sub></i> and <i>Cause<sub>RIPL</sub></i> fields: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td><i>IPL</i> and <i>RIPL</i> fields are 6-bits in width.</td></tr><tr><td>1</td><td><i>IPL</i> and <i>RIPL</i> fields are 8-bits in width.</td></tr><tr><td>Others</td><td>Reserved.</td></tr></table> <p>If the <i>IPL</i> field is 8-bits in width, bits 18 and 16 of <i>Status</i> are used as the most significant bit and second most significant bit, respectively, of that field.</p> <p>If the <i>RIPL</i> field is 8-bits in width, bits 17 and 16 of <i>Cause</i> are used as the most significant bit and second most significant bit, respectively, of that field.</p>	Encoding	Meaning	0	<i>IPL</i> and <i>RIPL</i> fields are 6-bits in width.	1	<i>IPL</i> and <i>RIPL</i> fields are 8-bits in width.	Others	Reserved.	R	Preset
Encoding	Meaning											
0	<i>IPL</i> and <i>RIPL</i> fields are 6-bits in width.											
1	<i>IPL</i> and <i>RIPL</i> fields are 8-bits in width.											
Others	Reserved.											
MMAR	20:18	microMIPS Architecture revision level: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Release 1</td></tr><tr><td>1-7</td><td>Reserved</td></tr></table>	Encoding	Meaning	0	Release 1	1-7	Reserved	R	Preset		
Encoding	Meaning											
0	Release 1											
1-7	Reserved											
MCU	17	MIPS MCU ASE implemented. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>MCU ASE is not implemented.</td></tr><tr><td>1</td><td>MCU ASE is implemented</td></tr></table>	Encoding	Meaning	0	MCU ASE is not implemented.	1	MCU ASE is implemented	R	Preset		
Encoding	Meaning											
0	MCU ASE is not implemented.											
1	MCU ASE is implemented											



Table 5.27 Config3 Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State										
Name	Bits													
ISAOncExc	16	Reflects the Instruction Set Architecture used when vectoring to an exception. Affects exceptions whose vectors are offsets from EBASE. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>MIPS32 ISA is used on entrance to an exception vector.</td></tr><tr><td>1</td><td>microMIPS is used on entrance to an exception vector.</td></tr></table>	Encoding	Meaning	0	MIPS32 ISA is used on entrance to an exception vector.	1	microMIPS is used on entrance to an exception vector.	RW	Preset, driven by signal external to CPU core				
Encoding	Meaning													
0	MIPS32 ISA is used on entrance to an exception vector.													
1	microMIPS is used on entrance to an exception vector.													
ISA	15:14	Indicates Instruction Set Availability. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Only MIPS32 is implemented.</td></tr><tr><td>1</td><td>Only microMIPS is implemented.</td></tr><tr><td>2</td><td>Both MIPS32 and MicroMIPS are implemented. MIPS32 ISA used when coming out of reset.</td></tr><tr><td>3</td><td>Both MIPS32 and MicroMIPS are implemented. MicroMIPS is used when coming out of reset.</td></tr></table>	Encoding	Meaning	0	Only MIPS32 is implemented.	1	Only microMIPS is implemented.	2	Both MIPS32 and MicroMIPS are implemented. MIPS32 ISA used when coming out of reset.	3	Both MIPS32 and MicroMIPS are implemented. MicroMIPS is used when coming out of reset.	R	Preset, driven by signal external to CPU core
Encoding	Meaning													
0	Only MIPS32 is implemented.													
1	Only microMIPS is implemented.													
2	Both MIPS32 and MicroMIPS are implemented. MIPS32 ISA used when coming out of reset.													
3	Both MIPS32 and MicroMIPS are implemented. MicroMIPS is used when coming out of reset.													
ULRI	13	<i>UserLocal</i> register implemented. This bit indicates whether the <i>UserLocal</i> coprocessor 0 register is implemented. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td><i>UserLocal</i> register is not implemented</td></tr><tr><td>1</td><td><i>UserLocal</i> register is implemented</td></tr></table>	Encoding	Meaning	0	<i>UserLocal</i> register is not implemented	1	<i>UserLocal</i> register is implemented	R	Preset				
Encoding	Meaning													
0	<i>UserLocal</i> register is not implemented													
1	<i>UserLocal</i> register is implemented													
RXI	12	Indicates whether the RIE and XIE bits exist within the <i>PageGrain</i> register.. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>The <i>RIE</i> and <i>XIE</i> bits are not implemented within the <i>PageGrain</i> register.</td></tr><tr><td>1</td><td>The <i>RIE</i> and <i>XIE</i> bits are implemented within the <i>PageGrain</i> register</td></tr></table>	Encoding	Meaning	0	The <i>RIE</i> and <i>XIE</i> bits are not implemented within the <i>PageGrain</i> register.	1	The <i>RIE</i> and <i>XIE</i> bits are implemented within the <i>PageGrain</i> register	R	0				
Encoding	Meaning													
0	The <i>RIE</i> and <i>XIE</i> bits are not implemented within the <i>PageGrain</i> register.													
1	The <i>RIE</i> and <i>XIE</i> bits are implemented within the <i>PageGrain</i> register													
ITL	8	Indicates that iFlowTrace hardware is present	R	Preset										

Table 5.27 Config3 Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State						
Name	Bits									
LPA	7	<p>Denotes the presence of support for large physical addresses on MIPS64 processors. Not used by MIPS32 processors and returns zero on read.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Large physical address support is not implemented</td></tr><tr><td>1</td><td>Large physical address support is implemented</td></tr></table> <p>For implementations of Release 1 of the Architecture, this bit returns zero on read.</p>	Encoding	Meaning	0	Large physical address support is not implemented	1	Large physical address support is implemented	R	Preset
Encoding	Meaning									
0	Large physical address support is not implemented									
1	Large physical address support is implemented									
VEIC	6	<p>Indicates support for an external interrupt controller.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Support for EIC interrupt mode is not implemented</td></tr><tr><td>1</td><td>Support for EIC interrupt mode is implemented</td></tr></table> <p>The value of this bit is set by the static input, <i>SI_EICPresent</i>. This allows external logic to communicate whether an external interrupt controller is attached to the processor or not.</p>	Encoding	Meaning	0	Support for EIC interrupt mode is not implemented	1	Support for EIC interrupt mode is implemented	R	Externally Set
Encoding	Meaning									
0	Support for EIC interrupt mode is not implemented									
1	Support for EIC interrupt mode is implemented									
VInt	5	<p>Indicates implementation of Vectored interrupts.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Vector interrupts are not implemented</td></tr><tr><td>1</td><td>Vectored interrupts are implemented</td></tr></table> <p>On the M14K core, this bit is always a 1, because vectored interrupts are implemented.</p>	Encoding	Meaning	0	Vector interrupts are not implemented	1	Vectored interrupts are implemented	R	1
Encoding	Meaning									
0	Vector interrupts are not implemented									
1	Vectored interrupts are implemented									
SP	4	<p>When set, indicates that Small (1KByte) page support is implemented.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Small page support is not implemented</td></tr><tr><td>1</td><td>Small page support is implemented</td></tr></table>	Encoding	Meaning	0	Small page support is not implemented	1	Small page support is implemented	R	0
Encoding	Meaning									
0	Small page support is not implemented									
1	Small page support is implemented									
CDMM	3	<p>Common Device Memory Map implemented. This bit indicates whether the CDMM is implemented.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>CDMM is not implemented</td></tr><tr><td>1</td><td>CDMM is implemented</td></tr></table>	Encoding	Meaning	0	CDMM is not implemented	1	CDMM is implemented	R	Preset
Encoding	Meaning									
0	CDMM is not implemented									
1	CDMM is implemented									

Table 5.27 Config3 Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State						
Name	Bits									
SM	1	SmartMIPS™ ASE implemented. This bit indicates whether the SmartMIPS ASE is implemented. Because SmartMIPS is not present on the M14K core, this bit will always be 0. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>SmartMIPS ASE is not implemented</td></tr><tr><td>1</td><td>SmartMIPS ASE is implemented</td></tr></table>	Encoding	Meaning	0	SmartMIPS ASE is not implemented	1	SmartMIPS ASE is implemented	R	0
Encoding	Meaning									
0	SmartMIPS ASE is not implemented									
1	SmartMIPS ASE is implemented									
TL	0	Trace Logic implemented. This bit indicates whether PC or data trace is implemented. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Trace logic is not implemented</td></tr><tr><td>1</td><td>Trace logic is implemented</td></tr></table>	Encoding	Meaning	0	Trace logic is not implemented	1	Trace logic is implemented	R	Preset
Encoding	Meaning									
0	Trace logic is not implemented									
1	Trace logic is implemented									

### 5.2.24 Configuration Register 4 (CP0 Register 16, Select 4)

**Compliance Level:** *Required* if any optional feature described by this register is implemented: Release 2 of the Architecture; *Optional* otherwise.

The *Config4* register encodes additional capabilities.

Figure 5-25 shows the format of the *Config4* register; Table 5.28 describes the *Config4* register fields.

Figure 5-25 Config4 Register Format

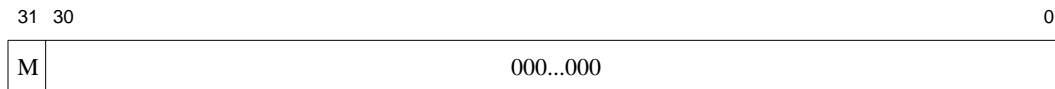


Table 5.28 Config4 Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
M	31	This bit is reserved to indicate that a <i>Config5</i> register is present.	R	1	Required
0	30:0	Must be written as zeros; returns zeros on read.	0	0	Reserved

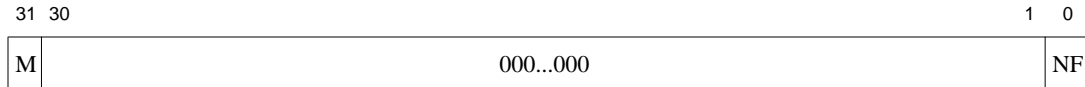
### 5.2.25 Config5 Register (CP0 Register 16, Select 5)

**Compliance Level:** *Required* if any optional feature described by this register is implemented: Release 2 of the Architecture; *Optional* otherwise.

The *Config5* register encodes additional capabilities.

Figure 5-26 shows the format of the *Config5* register; Table 5.29 describes the *Config5* register fields.

**Figure 5-26 Config5 Register Format**



**Table 5.29 Config5 Register Field Descriptions**

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
M	31	This bit is reserved to indicate that a <i>Config5</i> register is present. With the current architectural definition, this bit should always read as a 0.	R	0	Required
0	30:1	Must be written as zeros; returns zeros on read.	0	0	Reserved
NF Exist	0	Indicates that the Nested Fault feature is present. The Nested Fault feature allows recognition of faulting behavior within an exception handler.	R	1	Required

### 5.2.26 Config7 Register (CP0 Register 16, Select 7)

The *Config7* register contains implementation specific configuration information. A number of these bits are writeable to disable certain performance enhancing features within the M14K core.

**Figure 5.27 Config7 Register Format**



Table 5.30 Config7 Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
WII	31	Wait IE Ignore. Indicates that this processor will allow an interrupt to unblock a WAIT instruction, even if <i>/E</i> is preventing the interrupt from being taken. This avoids problems using the WAIT instruction for ‘bottom half’ interrupt servicing. In WII mode when <i>/E</i> =0, waking up from Sleep mode will not enter an Interrupt Service Routine.	R	1
0	30:19,17:0	These bits are unused and should be written as 0.	R	0
HCI	18	Hardware Cache Initialization: Indicates that a cache does not require initialization by software. This bit will most likely only be set on simulation-only cache models and not on real hardware.	R	0

### 5.2.27 Debug Register (CP0 Register 23, Select 0)

The *Debug* register is used to control the debug exception and provide information about the cause of the debug exception and also when re-entering at the debug exception vector due to a normal exception in debug mode. The read-only information bits are updated every time the debug exception is taken, or when a normal exception is taken when already in debug mode.

Only the *DM* bit and the *EJTAGver* field are valid when read from non-debug mode; the values of all other bits and fields are UNPREDICTABLE. Operation of the processor is UNDEFINED if the *Debug* register is written from non-debug mode.

Some of the bits and fields are only updated on debug exceptions and/or exceptions in debug mode, as shown below:

- *DSS*, *DBp*, *DDBL*, *DDBS*, *DIB*, *DINT*, *DIBImpr*, *DDBLImpr*, *DDBSImpr* are updated on both debug exceptions and on exceptions in debug modes.
- *DExcCode* is updated on exceptions in debug mode, and is undefined after a debug exception.
- *Halt* and *Doze* are updated on a debug exception, and are undefined after an exception in debug mode.
- *DBD* is updated on both debug and on exceptions in debug modes.

All bits and fields are undefined when read from normal mode, except those explicitly described to be defined, e.g., *EJTAGver* and *DM*.

Figure 5.28 Debug Register Format

31	30	29	28	27	26	25	24	23	22	21	20	19			
DBD	DM	NoDCR	LSNM	Doze	Halt	CountDM	IBusEP	MCheckP	CacheEP	DBusEP	IEXI	DDB-SImpr			
18	17	15	14	10	9	8	7	6	5	4	3	2	1	0	
DDBLI mpr	Ver		DExcCode			NoSSt	SSt	R	DIBI mpr	DINT	DIB	DDBS	DDBL	DBp	DSS

Table 5.31 Debug Register Field Descriptions

Fields		Description	Read/Write	Reset State						
Name	Bit(s)									
DBD	31	Indicates whether the last debug exception or exception in debug mode occurred in a branch delay slot: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Not in delay slot</td></tr><tr><td>1</td><td>In delay slot</td></tr></table>	Encoding	Meaning	0	Not in delay slot	1	In delay slot	R	Undefined
Encoding	Meaning									
0	Not in delay slot									
1	In delay slot									
DM	30	Indicates that the processor is operating in debug mode: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Processor is operating in non-debug mode</td></tr><tr><td>1</td><td>Processor is operating in debug mode</td></tr></table>	Encoding	Meaning	0	Processor is operating in non-debug mode	1	Processor is operating in debug mode	R	0
Encoding	Meaning									
0	Processor is operating in non-debug mode									
1	Processor is operating in debug mode									
NoDCR	29	Indicates whether the dseg memory segment is present and the Debug Control Register is accessible: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>dseg is present</td></tr><tr><td>1</td><td>No dseg present</td></tr></table>	Encoding	Meaning	0	dseg is present	1	No dseg present	R	0
Encoding	Meaning									
0	dseg is present									
1	No dseg present									
LSNM	28	Controls access of load/store between dseg and main memory: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Load/stores in dseg address range goes to dseg</td></tr><tr><td>1</td><td>Load/stores in dseg address range goes to main memory</td></tr></table>	Encoding	Meaning	0	Load/stores in dseg address range goes to dseg	1	Load/stores in dseg address range goes to main memory	R/W	0
Encoding	Meaning									
0	Load/stores in dseg address range goes to dseg									
1	Load/stores in dseg address range goes to main memory									
Doze	27	Indicates that the processor was in any kind of low power mode when a debug exception occurred: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Processor not in low-power mode when debug exception occurred</td></tr><tr><td>1</td><td>Processor in low-power mode when debug exception occurred</td></tr></table>	Encoding	Meaning	0	Processor not in low-power mode when debug exception occurred	1	Processor in low-power mode when debug exception occurred	R	Undefined
Encoding	Meaning									
0	Processor not in low-power mode when debug exception occurred									
1	Processor in low-power mode when debug exception occurred									
Halt	26	Indicates that the internal system bus clock was stopped when the debug exception occurred: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Internal system bus clock stopped</td></tr><tr><td>1</td><td>Internal system bus clock running</td></tr></table>	Encoding	Meaning	0	Internal system bus clock stopped	1	Internal system bus clock running	R	Undefined
Encoding	Meaning									
0	Internal system bus clock stopped									
1	Internal system bus clock running									

Table 5.31 Debug Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State	
Name	Bit(s)				
CountDM	25	Indicates the Count register behavior in debug mode:	R/W	1	
		Encoding			Meaning
		0			Count register stopped in debug mode
		1			Count register is running in debug mode
IBusEP	24	Instruction fetch Bus Error exception Pending. Set when an instruction fetch bus error event occurs, or if a 1 is written to the bit by software. Cleared when a Bus Error exception on an instruction fetch is taken by the processor, and by reset. If <i>IBusEP</i> is set when <i>IEXI</i> is cleared, a Bus Error exception on an instruction fetch is taken by the processor, and <i>IBusEP</i> is cleared.	R/W1	0	
MCheckP	23	Indicates that an imprecise Machine Check exception is pending. All Machine Check exceptions are precise on the M14K processor, so this bit will always read as 0.	R	0	
CacheEP	22	Indicates that an imprecise Cache Error is pending. Cache Errors cannot be taken by the M14K core, so this bit will always read as 0	R	0	
DBusEP	21	Data access Bus Error exception Pending. Covers imprecise bus errors on data access, similar to the behavior of <i>IBusEP</i> for imprecise bus errors on an instruction fetch.	R/W1	0	
IEXI	20	Imprecise Error eXception Inhibit controls exceptions taken due to imprecise error indications. Set when the processor takes a debug exception or exception in debug mode. Cleared by execution of the DERET instruction; otherwise modifiable by debug mode software. When <i>IEXI</i> is set, the imprecise error exception from a bus error on an instruction fetch or data access, cache error, or machine check is inhibited and deferred until the bit is cleared.	R/W	0	
DDBSImpr	19	Indicates that an imprecise Debug Data Break Store exception was taken. Imprecise data breaks only occur on complex breakpoints.	R	Undefined	
DDBLImpr	18	Indicates that an imprecise Debug Data Break Load exception was taken. Imprecise data breaks only occur on complex breakpoints.	R	Undefined	
Ver	17:15	EJTAG version.	R	101	
DExcCode	14:10	Indicates the cause of the latest exception in debug mode. The field is encoded as the <i>ExcCode</i> field in the <i>Cause</i> register for those normal exceptions that may occur in debug mode. Value is undefined after a debug exception.	R	Undefined	

Table 5.31 Debug Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State						
Name	Bit(s)									
NoSST	9	Indicates whether the single-step feature controllable by the <i>SSt</i> bit is available in this implementation: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Single-step feature available</td></tr><tr><td>1</td><td>No single-step feature available</td></tr></table>	Encoding	Meaning	0	Single-step feature available	1	No single-step feature available	R	0
Encoding	Meaning									
0	Single-step feature available									
1	No single-step feature available									
SSt	8	Controls if debug single step exception is enabled: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No debug single-step exception enabled</td></tr><tr><td>1</td><td>Debug single step exception enabled</td></tr></table>	Encoding	Meaning	0	No debug single-step exception enabled	1	Debug single step exception enabled	R/W	0
Encoding	Meaning									
0	No debug single-step exception enabled									
1	Debug single step exception enabled									
R	7	Reserved. Must be written as zeros; returns zeros on reads.	R	0						
DIBImpr	6	Indicates that an Imprecise debug instruction break exception occurred (due to a complex breakpoint). Cleared on exception in debug mode.	R	Undefined						
DINT	5	Indicates that a debug interrupt exception occurred. Cleared on exception in debug mode. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No debug interrupt exception</td></tr><tr><td>1</td><td>Debug interrupt exception</td></tr></table>	Encoding	Meaning	0	No debug interrupt exception	1	Debug interrupt exception	R	Undefined
Encoding	Meaning									
0	No debug interrupt exception									
1	Debug interrupt exception									
DIB	4	Indicates that a debug instruction break exception occurred. Cleared on exception in debug mode. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No debug instruction exception</td></tr><tr><td>1</td><td>Debug instruction exception</td></tr></table>	Encoding	Meaning	0	No debug instruction exception	1	Debug instruction exception	R	Undefined
Encoding	Meaning									
0	No debug instruction exception									
1	Debug instruction exception									
DDBS	3	Indicates that a debug data break exception occurred on a store. Cleared on exception in debug mode. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No debug data exception on a store</td></tr><tr><td>1</td><td>Debug instruction exception on a store</td></tr></table>	Encoding	Meaning	0	No debug data exception on a store	1	Debug instruction exception on a store	R	Undefined
Encoding	Meaning									
0	No debug data exception on a store									
1	Debug instruction exception on a store									
DDBL	2	Indicates that a debug data break exception occurred on a load. Cleared on exception in debug mode. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No debug data exception on a load</td></tr><tr><td>1</td><td>Debug instruction exception on a load</td></tr></table>	Encoding	Meaning	0	No debug data exception on a load	1	Debug instruction exception on a load	R	Undefined
Encoding	Meaning									
0	No debug data exception on a load									
1	Debug instruction exception on a load									



Table 5.31 Debug Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State						
Name	Bit(s)									
DBp	1	Indicates that a debug software breakpoint exception occurred. Cleared on exception in debug mode.	R	Undefined						
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No debug software breakpoint exception</td></tr><tr><td>1</td><td>Debug software breakpoint exception</td></tr></table>			Encoding	Meaning	0	No debug software breakpoint exception	1	Debug software breakpoint exception
		Encoding			Meaning					
		0			No debug software breakpoint exception					
1	Debug software breakpoint exception									
DSS	0	Indicates that a debug single-step exception occurred. Cleared on exception in debug mode.	R	Undefined						
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No debug single-step exception</td></tr><tr><td>1</td><td>Debug single-step exception</td></tr></table>			Encoding	Meaning	0	No debug single-step exception	1	Debug single-step exception
Encoding	Meaning									
0	No debug single-step exception									
1	Debug single-step exception									

### 5.2.28 Trace Control Register (CP0 Register 23, Select 1)

The *TraceControl* register configuration is shown below.

This register is only implemented if the EJTAG PDTrace capability is present.

Figure 5.29 TraceControl Register Format

31	30	29	28	27	26	25	24	23	22	21	20	13				12	5			4	3	1		0
TS	UT	0		TB	IO	D	E	K	S	U	ASID_M				ASID				G	Mode		On		

Table 5.32 TraceControl Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
TS	31	The trace select bit is used to select between the hardware and the software trace control bits. A value of zero selects the external hardware trace block signals, and a value of one selects the trace control bits in this software control register.	R/W	0
UT	30	This bit is used to indicate the type of user-triggered trace record. A value of zero implies a user type 1, and a value of one implies a user type 2. The actual triggering of a user trace record occurs on a write to the <i>UserTraceData</i> register.	R/W	Undefined
0	29:28	Reserved for future use; Must be written as zero; returns zero on read.	0	0

**Table 5.32 TraceControl Register Field Descriptions (Continued)**

Fields		Description	Read/Write	Reset State
Name	Bits			
TB	27	Trace All Branch. When set to one, this tells the processor to trace the PC value for all taken branches, not just the ones whose branch target address is statically unpredictable.	R/W	Undefined
IO	26	Inhibit Overflow. This signal is used to indicate to the core trace logic that slow but complete tracing is desired. When set to one, the core tracing logic does not allow a FIFO overflow or discard trace data. This is achieved by stalling the pipeline when the FIFO is nearly full, so that no trace records are ever lost.	R/W	Undefined
D	25	When set to one, this enables tracing in Debug Mode. For trace to be enabled in Debug mode, the On bit must be one. When set to zero, trace is disabled in Debug Mode, irrespective of other bits.	R/W	Undefined
E	24	When set to one, this enables tracing in Exception Mode. For trace to be enabled in Exception mode, the On bit must be one. When set to zero, trace is disabled in Exception Mode, irrespective of other bits.	R/W	Undefined
K	23	When set to one, this enables tracing in Kernel Mode. For trace to be enabled in Kernel mode, the On bit must be one. When set to zero, trace is disabled in Kernel Mode, irrespective of other bits.	R/W	Undefined
0	22	This bit is reserved. Must be written as zero; returns zero on read.	0	0
U	21	When set to one, this enables tracing in User Mode. For trace to be enabled in User mode, the On bit must be one. When set to zero, trace is disabled in User Mode, irrespective of other bits.	R/W	Undefined
ASID_M	20:13	In an FM-based MMU core in which ASID is not supported, this field is ignored on writes and returns zero on reads.	R	0
ASID	12:5	In an FM-based MMU core in which ASID is not supported, this field is ignored on writes and returns zero on reads.	R	0
G	4	In an FM-based MMU core in which ASID is not supported, this field is ignored on writes and returns 1 on reads. This causes all match equations to work correctly in the absence of an ASID.	R	1

Table 5.32 TraceControl Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State																		
Name	Bits																					
Mode	3:1	<p>These three bits control the trace mode function.</p> <table><thead><tr><th>Mode</th><th>Trace Mode</th></tr></thead><tbody><tr><td>000</td><td>Trace PC</td></tr><tr><td>001</td><td>Trace PC and load address</td></tr><tr><td>010</td><td>Trace PC and store address</td></tr><tr><td>011</td><td>Trace PC and both load/store addresses</td></tr><tr><td>100</td><td>Trace PC and load data</td></tr><tr><td>101</td><td>Trace PC and load address and data</td></tr><tr><td>110</td><td>Trace PC and store address and data</td></tr><tr><td>111</td><td>Trace PC and both load/store address and data</td></tr></tbody></table> <p>The <i>TraceControl2</i><sub>ValidModes</sub> field determines which of these encodings are supported by the processor. The operation of the processor is <b>UNPREDICTABLE</b> if this field is set to a value which is not supported by the processor.</p>	Mode	Trace Mode	000	Trace PC	001	Trace PC and load address	010	Trace PC and store address	011	Trace PC and both load/store addresses	100	Trace PC and load data	101	Trace PC and load address and data	110	Trace PC and store address and data	111	Trace PC and both load/store address and data	R/W	Undefined
Mode	Trace Mode																					
000	Trace PC																					
001	Trace PC and load address																					
010	Trace PC and store address																					
011	Trace PC and both load/store addresses																					
100	Trace PC and load data																					
101	Trace PC and load address and data																					
110	Trace PC and store address and data																					
111	Trace PC and both load/store address and data																					
On	0	This is the master trace enable switch in software control. When zero, tracing is always disabled. When set to one, tracing is enabled whenever the other enabling functions are also true.	R/W	0																		

### 5.2.29 Trace Control2 Register (CP0 Register 23, Select 2)

The *TraceControl2* register provides additional control and status information. Note that some fields in the *TraceControl2* register are read-only, but have a reset state of “Undefined”. This is because these values are loaded from the Trace Control Block (TCB) (see [Section 8.8.6 “ITCB Register Interface for Software Configurability”](#)). As such, these fields in the *TraceControl2* register will not have valid values until the TCB asserts these values.

This register is only implemented if the EJTAG PDTrace capability is present.

Figure 5.30 TraceControl2 Register Format

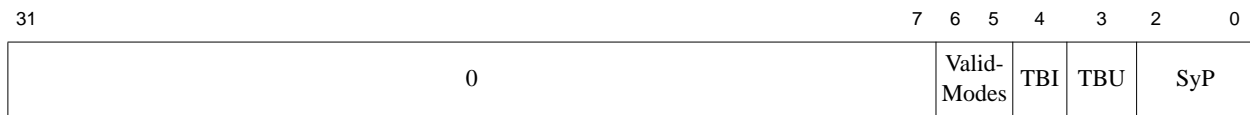


Table 5.33 TraceControl2 Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:5	Reserved for future use; Must be written as zero; returns zero on read.	0	0

Table 5.33 TraceControl2 Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State										
Name	Bits													
ValidModes	6:5	<p>This field specifies the type of tracing that is supported by the processor.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>00</td><td>PC tracing only</td></tr><tr><td>01</td><td>PC and load and store address tracing only</td></tr><tr><td>10</td><td>PC, load and store address, and load and store data</td></tr><tr><td>11</td><td>Reserved</td></tr></table>	Encoding	Meaning	00	PC tracing only	01	PC and load and store address tracing only	10	PC, load and store address, and load and store data	11	Reserved	R	10
			Encoding	Meaning										
			00	PC tracing only										
			01	PC and load and store address tracing only										
			10	PC, load and store address, and load and store data										
			11	Reserved										
TBI	4	<p>This bit indicates how many trace buffers are implemented by the TCB.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Only one trace buffer is implemented, and the Debug single-step exception bit of this register indicates which trace buffer is implemented</td></tr><tr><td>1</td><td>Both on-chip and off-chip trace buffers are implemented by the TCB, and the <i>TBU</i> bit of this register indicates to which trace buffer the trace is currently written.</td></tr></table>	Encoding	Meaning	0	Only one trace buffer is implemented, and the Debug single-step exception bit of this register indicates which trace buffer is implemented	1	Both on-chip and off-chip trace buffers are implemented by the TCB, and the <i>TBU</i> bit of this register indicates to which trace buffer the trace is currently written.	R	Per implementation				
			Encoding	Meaning										
			0	Only one trace buffer is implemented, and the Debug single-step exception bit of this register indicates which trace buffer is implemented										
			1	Both on-chip and off-chip trace buffers are implemented by the TCB, and the <i>TBU</i> bit of this register indicates to which trace buffer the trace is currently written.										
TBU	3	<p>This bit denotes to which trace buffer the trace is currently being written and is used to select the appropriate interpretation of the <i>TraceControl2</i><sub>SYP</sub> field.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Trace data is being sent to an on-chip trace buffer</td></tr><tr><td>1</td><td>Trace Data is being sent to an off-chip trace buffer</td></tr></table>	Encoding	Meaning	0	Trace data is being sent to an on-chip trace buffer	1	Trace Data is being sent to an off-chip trace buffer	R	Undefined				
			Encoding	Meaning										
			0	Trace data is being sent to an on-chip trace buffer										
			1	Trace Data is being sent to an off-chip trace buffer										

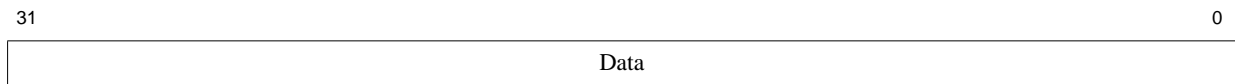
**Table 5.33 TraceControl2 Register Field Descriptions (Continued)**

Fields		Description	Read/Write	Reset State																											
Name	Bits																														
SyP	2:0	<p>Used to indicate the synchronization period. The period (in cycles) between which the periodic synchronization information is to be sent is defined as shown below, for both when the trace buffer is on-chip and off-chip.</p> <table><tr><th>SyP</th><th>On-chip</th><th>Off-chip</th></tr><tr><td>000</td><td><math>2^2</math></td><td><math>2^7</math></td></tr><tr><td>001</td><td><math>2^3</math></td><td><math>2^8</math></td></tr><tr><td>010</td><td><math>2^4</math></td><td><math>2^9</math></td></tr><tr><td>011</td><td><math>2^5</math></td><td><math>2^{10}</math></td></tr><tr><td>100</td><td><math>2^6</math></td><td><math>2^{11}</math></td></tr><tr><td>101</td><td><math>2^7</math></td><td><math>2^{12}</math></td></tr><tr><td>110</td><td><math>2^8</math></td><td><math>2^{13}</math></td></tr><tr><td>111</td><td><math>2^9</math></td><td><math>2^{14}</math></td></tr></table> <p>The “On-chip” column value is used when the trace data is being written to an on-chip trace buffer (e.g, <i>TraceControl2</i><sub>TBU</sub> = 0). Conversely, the “Off-chip” column is used when the trace data is being written to an off-chip trace buffer (e.g, <i>TraceControl2</i><sub>TBU</sub> = 1).</p>	SyP	On-chip	Off-chip	000	$2^2$	$2^7$	001	$2^3$	$2^8$	010	$2^4$	$2^9$	011	$2^5$	$2^{10}$	100	$2^6$	$2^{11}$	101	$2^7$	$2^{12}$	110	$2^8$	$2^{13}$	111	$2^9$	$2^{14}$	R	Undefined
SyP	On-chip	Off-chip																													
000	$2^2$	$2^7$																													
001	$2^3$	$2^8$																													
010	$2^4$	$2^9$																													
011	$2^5$	$2^{10}$																													
100	$2^6$	$2^{11}$																													
101	$2^7$	$2^{12}$																													
110	$2^8$	$2^{13}$																													
111	$2^9$	$2^{14}$																													

### 5.2.30 User Trace Data1 Register (CP0 Register 23, Select 3)/User Trace Data2 Register (CP0 Register 24, Select 3)

A software write to any bits in the *UserTraceData1* or *UserTraceData2* registers will trigger a trace record to be written indicating a type 1 or type 2 user format respectively. The trace output data is **UNPREDICTABLE** if these registers are written in consecutive cycles.

This register is only implemented if the MIPS iFlowtrace capability is present.

**Figure 5.31 User Trace Data1/User Trace Data2 Register Format****Table 5.34 UserTraceData1/UserTraceData2 Register Field Descriptions**

Fields		Description	Read / Write	Reset State
Name	Bits			
Data	31:0	Software readable/writable data. When written, this triggers a user format trace record out of the PDtrace interface that transmits the Data field to trace memory.	R/W	0

### 5.2.31 TraceBPC Register (CP0 Register 23, Select 4)

This register is used to start and stop tracing using an EJTAG Hardware breakpoint. The Hardware breakpoint can then be set as a trigger source and optionally also as a Debug exception breakpoint.

This register is only implemented if hardware breakpoints and the EJTAG PDTrace capability are both present.

**Figure 5.32 Trace BPC Register Format**

31	30		18	17	16	15	14		6	5		0
DE		0			DBPOn	IE		0			IBPOn	

**Table 5.35 TraceBPC Register Field Descriptions**

Fields		Description	Read/Write	Reset State						
Name	Bits									
DE	31	Used to specify whether the trigger signal from EJTAG data breakpoint should trigger tracing functions or not: 0: disables trigger signals from data breakpoints 1: enables trigger signals from data breakpoints	R/W	0						
0	30:18	Reserved	0	0						
DBPOn	17:16	Each of the 2 bits corresponds to the 2 possible EJTAG hardware data breakpoints that may be implemented. For example, bit 16 corresponds to the first data breakpoint. If 2 data breakpoints are present in the EJTAG implementation, then they correspond to bits 16 and 17. The rest are always ignored by the tracing logic because they will never be triggered. A value of one for each bit implies that a trigger from the corresponding data breakpoint should start tracing. And a value of zero implies that tracing should be turned off with the trigger signal.	R/W	0						
IE	15	Used to specify whether the trigger signal from EJTAG instruction breakpoint should trigger tracing functions or not: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Disables trigger signals from instruction breakpoints</td></tr><tr><td>1</td><td>Enables trigger signals from instruction breakpoints</td></tr></table>	Encoding	Meaning	0	Disables trigger signals from instruction breakpoints	1	Enables trigger signals from instruction breakpoints	R/W	0
Encoding	Meaning									
0	Disables trigger signals from instruction breakpoints									
1	Enables trigger signals from instruction breakpoints									
0	14:6	Reserved	0	0						

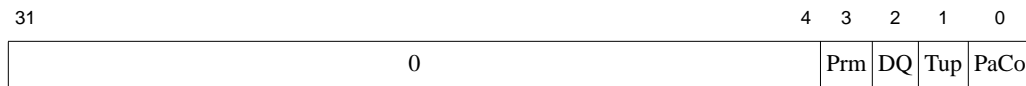
**Table 5.35 TraceBPC Register Field Descriptions (Continued)**

Fields		Description	Read/Write	Reset State
Name	Bits			
IBPOn	5:0	Each of the 6 bits corresponds to the 6 possible EJTAG hardware instruction breakpoints that may be implemented. Bit 0 corresponds to the first instruction breakpoint, and so on. If only 2 instruction breakpoints are present in the EJTAG implementation, then only bits 0 and 1 are used. The rest are always ignored by the tracing logic because they will never be triggered. A value of one for each bit implies that a trigger from the corresponding instruction breakpoint should start tracing. And a value of zero implies that tracing should be turned off with the trigger signal.	R/W	0

### 5.2.32 Debug2 Register (CP0 Register 23, Select 6)

This register holds additional information about Complex Breakpoint exceptions.

This register is only implemented if complex hardware breakpoints are present.

**Figure 5.33 Debug2 Register Format****Table 5.36 Debug2 Register Field Descriptions**

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:4	Reserved	0	0
Prm	3	Primed - indicates whether a complex breakpoint with an active priming condition was seen on the last debug exception.	R	Undefined
DQ	2	Data Qualified - indicates whether a complex breakpoint with an active data qualifier was seen on the last debug exception.	R	Undefined
Tup	1	Tuple - indicates whether a tuple breakpoint was seen on the last debug exception.	R	Undefined
PaCo	0	Pass Counter - indicates whether a complex breakpoint with an active pass counter was seen on the last debug exception	R	Undefined

### 5.2.33 Debug Exception Program Counter Register (CP0 Register 24, Select 0)

The Debug Exception Program Counter (*DEPC*) register is a read/write register that contains the address at which processing resumes after a debug exception or debug mode exception has been serviced.

For synchronous (precise) debug and debug mode exceptions, the *DEPC* contains either:

- The virtual address of the instruction that was the direct cause of the debug exception, or
- The virtual address of the immediately preceding branch or jump instruction, when the debug exception causing instruction is in a branch delay slot, and the *Debug Branch Delay (DBD)* bit in the *Debug* register is set.

For asynchronous debug exceptions (debug interrupt, complex break), the *DEPC* contains the virtual address of the instruction where execution should resume after the debug handler code is executed.

In processors that implement microMIPS, a read of the *DEPC* register (via MFC0) returns the following value in the destination GPR:

$$\text{GPR}[\text{rt}] \leftarrow \text{DebugExceptionPC}_{31..1} \parallel \text{ISAMode}_0$$

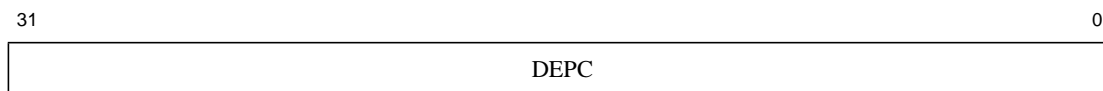
That is, the upper 31 bits of the debug exception PC are combined with the lower bit of the *ISAMode* field and written to the GPR.

Similarly, a write to the *DEPC* register (via MTC0) takes the value from the GPR and distributes that value to the debug exception PC and the *ISAMode* field, as follows

$$\begin{aligned} \text{DebugExceptionPC} &\leftarrow \text{GPR}[\text{rt}]_{31..1} \parallel 0 \\ \text{ISAMode} &\leftarrow 2\#0 \parallel \text{GPR}[\text{rt}]_0 \end{aligned}$$

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the debug exception PC, and the lower bit of the debug exception PC is cleared. The upper bit of the *ISAMode* field is cleared and the lower bit is loaded from the lower bit of the GPR.

**Figure 5.34 DEPC Register Format**



**Table 5.37 DEPC Register Formats**

Fields		Description	Read/Write	Reset
Name	Bit(s)			
DEPC	31:0	The <i>DEPC</i> register is updated with the virtual address of the instruction that caused the debug exception. If the instruction is in the branch delay slot, then the virtual address of the immediately preceding branch or jump instruction is placed in this register. Execution of the DERET instruction causes a jump to the address in the <i>DEPC</i> .	R/W	Undefined



### 5.2.34 Performance Counter Register (CP0 Register 25, select 0-3)

The M14K processor defines two performance counters and two associated control registers, which are mapped to CP0 register 25. The select field of the MTC0/MFC0 instructions are used to select the specific register accessed by the instruction, as shown in [Table 5.38](#).

### Table 5.38 Performance Counter Register Selects

Select[2:0]	Register
0	Register 0 Control
1	Register 0 Count
2	Register 1 Control
3	Register 1 Count

Each counter is a 32-bit read/write register and is incremented by one each time the countable event, specified in its associated control register, occurs. Each counter can independently count one type of event at a time.

Bit 31 of each of the counters are ANDed with an interrupt enable bit, *IE*, of their respective control register to determine if a performance counter interrupt should be signalled. The two values are then OR'ed together to create the *SI\_PCI* output. This signal is combined with one of the *SI\_Int* pins to signal an interrupt to the M14K. Counting is not affected by the interrupt indication. This output is cleared when the counter wraps to zero, and may be cleared in software by writing a value with bit 31 = 0 to the *Performance Counter Count* registers.

NOTE: the performance counter registers are connected to a clock that is stopped when the processor is in sleep mode (if the top level clock gater is present). Most events would not be active during that time, but others would be, notably the cycle count. This behavior should be considered when analyzing measurements taken on a system. Further, note that FPGA implementations of the core would generally not have the clock gater present and thus would have different behavior than a typical ASIC implementation.

### Figure 5.35 Performance Counter Control Register

31 30		12 11		5 4 3 2 1 0	
M	0	Event	IE	U	0 K EXL

### Table 5.39 Performance Counter Control Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
M	31	If this bit is one, another pair of <i>Performance Control</i> and <i>Counter</i> registers is implemented at a MTC0 or MFC0 select field value of ‘n+2’ and ‘n+3’.	R	Preset
Event	11:5	Counter event enabled for this counter. Possible events are listed in <a href="#">Table 6.60</a> .	R/W	Undefined
IE	4	Counter Interrupt Enable. This bit masks bit 31 of the associated count register from the interrupt exception request output.	R/W	0
U	3	Count in User Mode. When this bit is set, the specified event is counted in User Mode.	R/W	Undefined
K	1	Count in Kernel Mode. When this bit is set, count the event in Kernel Mode when <i>EXL</i> and <i>ERL</i> both are 0.	R/W	Undefined

Table 5.39 Performance Counter Control Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
EXL	0	Count when <i>EXL</i> . When this bit is set, count the event when <i>EXL</i> = 1 and <i>ERL</i> = 0.	R/W	Undefined
0	30:12, 2	Must be written as zeroes; returns zeroes when read.	0	0

Table 6.60 describes the events countable with the two performance counters. The mode column indicates whether the event counting is influenced by the mode bits (*U,K,EXL*). The operation of a counter is **UNPREDICTABLE** for events which are specified as Reserved.

Performance counters never count in debug mode or when *ERL* = 1.

Event Num	Counter 0	Mode	Counter 1	Mode
0	Cycles	No	Cycles	No
1	Instructions completed	Yes	Instructions completed	Yes
2	branch instructions	Yes	Reserved	NA
3	JR r31 (return) instructions	Yes	Reserved	NA
4	JR (not r31) instructions	Yes	Reserved	NA
5	Reserved	NA	Reserved	NA
6	Reserved	NA	Reserved	NA
7	Reserved	NA	Reserved	NA
8	Reserved	NA	Reserved	NA
9	Reserved	NA	Reserved	NA
10	Reserved	NA	Reserved	NA
11	Reserved	NA	Reserved	NA
12	Reserved	NA	Reserved	NA
13	Reserved	NA	Reserved	NA
14	integer instructions completed	Yes	Reserved	NA
15	loads completed	Yes	Stores completed	Yes
16	J/JAL completed	Yes	microMIPS instructions completed	Yes
17	no-ops completed	Yes	Integer multiply/divide completed	Yes
18	Stall cycles	No	Reserved	NA
19	SC instructions completed	Yes	SC instructions failed	Yes
20	Prefetch instructions completed	Yes	Reserved	NA

## 5.2 CP0 Register Descriptions

Event Num	Counter 0	Mode	Counter 1	Mode
21	Reserved	NA	Reserved	NA
22	Reserved	NA	Reserved	NA
23	Exceptions taken	Yes	Reserved	NA
24	Reserved	NA	Reserved	NA
25	Reserved	NA	ALU stall cycles	No
26	Reserved	NA	Reserved	NA
27	Reserved	NA	Reserved	NA
28	Reserved	NA	Implementation-specific CP2 event	Yes
29	Reserved	NA	Reserved	NA
30	Implementation-specific CorExtend event	Yes	Reserved	NA
31	Reserved	NA	Reserved	NA
32	Reserved	NA	Reserved	NA
33	Reserved	NA	Reserved	NA
34	Reserved	NA	Reserved	NA
35	Reserved	NA	CP2 To/From Instructions completed	Yes
36	Reserved			
37	Reserved	NA	Reserved	NA
38	Reserved	NA	Reserved	NA
39	Reserved	NA	Reserved	NA
40	Uncached stall cycles	Yes	Reserved	NA
41	MDU stall cycles	Yes	Reserved	NA
42	CP2 stall cycles	Yes	CorExtend stall cycles	Yes
43	Reserved	NA	Reserved	NA
44	Reserved	NA	Reserved	NA
45	Load to Use stall cycles	Yes	Reserved	NA
46	Other interlock stall cycles	Yes	Reserved	NA
47	Reserved	NA	Reserved	NA
48	Reserved	NA	Reserved	NA
49	EJTAG Instruction Triggerpoints	Yes	EJTAG Data Triggerpoints	Yes
50	Reserved	NA	Reserved	NA
51	Reserved	NA	Reserved	NA
52	Reserved	NA	Reserved	NA

Event Num	Counter 0	Mode	Counter 1	Mode
53	Reserved	NA	Reserved	NA
54	Reserved	NA	Reserved	NA
55	Reserved	NA	Reserved	NA
56-63	Reserved	NA	Reserved	NA

Table 5.40 Event Descriptions

Event Name	Counter	Event Number	Description
Cycles	0/1	0	Total number of cycles. The performance counters are clocked by the top-level gated clock. If the M14K is built with that clock gater present, none of the counters will increment while the clock is stopped, e.g., due to a WAIT instruction.
<b>Instruction Completion:</b> The following events indicate completion of various types of instructions			
Instructions	0/1	1	Total number of instructions completed.
Branch instns	0	2	Counts all branch instructions that completed.
JR R31 (return) instns	0	3	Counts all JR R31 instructions that completed.
JR (not R31)	0	4	Counts all JR \$xx (not \$31) and JALR instructions (indirect jumps).
Integer instns	0	14	Non-floating point, non-Coprocessor 2 instructions.
Loads	0	15	Includes both integer and coprocessor loads.
Stores	1	15	Includes both integer and coprocessor stores.
J/JAL	0	16	Direct Jump (And Link) instruction.
microMIPS	1	16	All microMIPS instructions.
no-ops	0	17	This includes all instructions that normally write to a GPR, but where the destination register was set to r0.
Integer Multiply/Divide	1	17	Counts all Integer Multiply/Divide instructions (MULxx, DIVx, MADDx, MSUBx).
SC	0	19	Counts conditional stores regardless of whether they succeeded.
PREF	0	20	Note that this only counts PREFs that are actually attempted. PREFs to uncached addresses or ones with translation errors are not counted
Cp2 To/From instns	1	35	Includes move to/from, control to/from, and cop2 loads and stores.
<b>Instruction execution events</b>			
SC instructions failed	1	19	SC instruction that did not update memory. Note: While this event and the SC instruction count event can be configured to count in specific operating modes, the timing of the events is much different, and the observed operating mode could change between them, causing some inaccuracy in the measured ratio.
Exceptions Taken	0	23	Any type of exception taken.
EJTAG instruction triggers	0	49	Number of times an EJTAG Instruction Trigger Point condition matched.
EJTAG data triggers	1	49	Number of times an EJTAG Data Trigger Point condition matched.

Table 5.40 Event Descriptions (Continued)

Event Name	Counter	Event Number	Description
<b>General Stalls</b>			
ALU stall cycles	1	25	Counts the number of cycles in which the ALU pipeline cannot advance.
Stall cycles	0	18	Counts the total number of cycles in which no instructions are issued by SRAM to ALU (the RF stage does not advance). This includes both of the previous two events. However, this is different from the sum of them, because cycles when both stalls are active will only be counted once.
<b>Specific stalls</b> - these events will count the number of cycles lost due to this. This will include bubbles introduced by replays within the pipe. If multiple stall sources are active simultaneously, the counters for each of the active events will be incremented.			
Uncached stall cycles	0	40	Cycles in which the processor is stalled on an uncached fetch, load, or store.
MDU stall cycles	0	41	Counts all cycles in which the integer pipeline waits on MDU return data.
Cp2 stall cycles	0	42	Counts all cycles in which the integer pipeline waits on CP2 return data.
CorExtend stall cycles	1	42	Counts all cycles in which the integer pipeline waits on CorExtend return data.
Load to Use stall cycles	0	45	Counts all cycles in which the integer pipeline waits on Load return data.
Other interlocks stall cycles	0	46	Counts all cycles in which the integer pipeline waits on return data from MFC0 and RDHWR instructions.
<b>Implementation specific events</b> - Modules that can be replaced by the customer will have an event signal associated with them.			
Cp2	1	28	Set to 1 if COP2 is implemented.
CorExtend	0	30	Set to 1 if CorExtend is implemented.

The performance counter resets to a low-power state, in which none of the counters will start counting events until software has enabled event counting, using an MTC0 instruction to the Performance Counter Control Registers.

Figure 5.36 Performance Counter Count Register

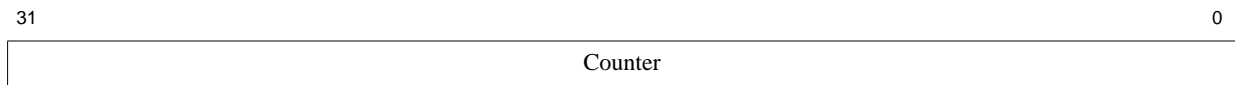


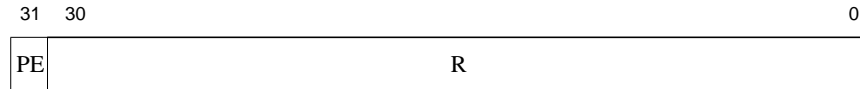
Table 5.41 Performance Counter Count Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Counter	31:0	Counter	R/W	Undefined

### 5.2.35 ErrCtl Register (CP0 Register 26, Select 0)

The *ErrCtl* register controls parity protection of data and instruction SRAM. Parity protection can be enabled or disabled using the *PE* bit.

**Figure 5.37 ErrCtl Register Format**



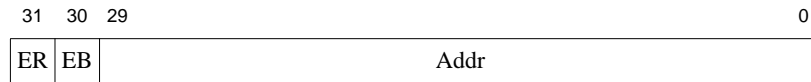
**Table 5.42 ErrCtl Register Field Descriptions**

Fields		Description	Read/Write	Reset State						
Name	Bit(s)									
PE	31	<p>Parity Enable. This bit enables or disables the parity protection for both the instruction SRAM and the data SRAM.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Parity disabled</td></tr><tr><td>1</td><td>Parity enabled</td></tr></table> <p>This field is only write-able if the parity option was implemented when the M14K was built. If parity is not supported, this field is always read as 0. Software can test for parity support by attempting to write a 1 to this field, then read back the value.</p>	Encoding	Meaning	0	Parity disabled	1	Parity enabled	R or R/W	0
Encoding	Meaning									
0	Parity disabled									
1	Parity enabled									
R	30:0	Must be written as zero; returns zero on reads.	0	0						

### 5.2.36 CacheErr Register (CP0 Register 27, Select 0)

The *CacheErr* register provides an interface with the cache error-detection logic. When a SRAM Parity Error exception is signaled, the fields of this register are set accordingly.

**Figure 5.38 CacheErr Register (Primary Caches)**



**Table 5.43 CacheErr Register Field Descriptions (Primary Caches)**

Fields		Description	Read / Write	Reset State						
Name	Bits									
ER	31	Error Reference. Indicates the type of reference that encountered an error.	R	Undefined						
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Instruction</td></tr><tr><td>1</td><td>Data</td></tr></table>			Encoding	Meaning	0	Instruction	1	Data
		Encoding			Meaning					
		0			Instruction					
		1			Data					

Table 5.43 CacheErr Register Field Descriptions (Primary Caches) (Continued)

Fields		Description	Read / Write	Reset State						
Name	Bits									
EB	30	Error Both. Indicates that a data SRAM parity error occurred in addition to an instruction SRAM parity error.	R	Undefined						
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No additional data SRAM parity error</td></tr><tr><td>1</td><td>Additional data SRam parity error</td></tr></table>			Encoding	Meaning	0	No additional data SRAM parity error	1	Additional data SRam parity error
		Encoding			Meaning					
		0			No additional data SRAM parity error					
		1			Additional data SRam parity error					
	In the case of an additional data SRAM parity error, the remainder of the bits in this register are set according to the instruction SRAM parity error.									
Addr	29:0	Error address. Specifies on which address the error was detected.	R	Undefined						

### 5.2.37 ErrorEPC (CP0 Register 30, Select 0)

The *ErrorEPC* register is a read/write register, similar to the *EPC* register, except that *ErrorEPC* is used on error exceptions. All bits of the *ErrorEPC* register are significant and must be writable. It is also used to store the program counter on Reset, Soft Reset, and nonmaskable interrupt (NMI) exceptions.

The *ErrorEPC* register contains the virtual address at which instruction processing can resume after servicing an error. This address can be:

- The virtual address of the instruction that caused the exception
- The virtual address of the immediately preceding branch or jump instruction when the error causing instruction is in a branch delay slot

Unlike the *EPC* register, there is no corresponding branch delay slot indication for the *ErrorEPC* register.

In processors that implement microMIPS, a read of the *ErrorEPC* register (via MFC0) returns the following value in the destination GPR:

$$\text{GPR}[\text{rt}] \leftarrow \text{ErrorExceptionPC}_{31..1} \parallel \text{ISAMode}_0$$

That is, the upper 31 bits of the error exception PC are combined with the lower bit of the *ISAMode* field and written to the GPR.

Similarly, a write to the *ErrorEPC* register (via MTC0) takes the value from the GPR and distributes that value to the error exception PC and the *ISAMode* field, as follows

$$\begin{aligned} \text{ErrprExceptionPC} &\leftarrow \text{GPR}[\text{rt}]_{31..1} \parallel 0 \\ \text{ISAMode} &\leftarrow 2\#0 \parallel \text{GPR}[\text{rt}]_0 \end{aligned}$$

That is, the upper 31 bits of the GPR are written to the upper 31 bits of the error exception PC, and the lower bit of the error exception PC is cleared. The upper bit of the *ISAMode* field is cleared and the lower bit is loaded from the lower bit of the GPR.

Figure 5.39 ErrorEPC Register Format



Table 5.44 ErrorEPC Register Field Description

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
ErrorEPC	31:0	Error Exception Program Counter.	R/W	Undefined

### 5.2.38 DeSave Register (CP0 Register 31, Select 0)

The *Debug Exception Save (DeSave)* register is a read/write register that functions as a simple memory location. This register is used by the debug exception handler to save one of the GPRs that is then used to save the rest of the context to a pre-determined memory area (such as in the EJTAG Probe). This register allows the safe debugging of exception handlers and other types of code in which the existence of a valid stack for context saving cannot be assumed.

Figure 5.40 DeSave Register Format



Table 5.45 DeSave Register Field Description

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
DESAVE	31:0	Debug exception save contents.	R/W	Undefined



## Hardware and Software Initialization of the M14K™ Core

An M14K processor core contains only a minimal amount of hardware initialization and relies on software to fully initialize the device.

This chapter contains the following sections:

- [Section 6.1 “Hardware-Initialized Processor State”](#)
- [Section 6.2 “Software Initialized Processor State”](#)

### 6.1 Hardware-Initialized Processor State

An M14K processor core, like most other MIPS processors, is not fully initialized by hardware reset. Only a minimal subset of the processor state is cleared. This is enough to bring the core up while running in unmapped and uncached code space. All other processor state can then be initialized by software. *SI\_ColdReset* is asserted after power-up to bring the device into a known state. Soft reset can be forced by asserting the *SI\_Reset* pin. This distinction is made for compatibility with other MIPS processors. In practice, both resets are handled identically with the exception of the setting of *Status<sub>SR</sub>*.

#### 6.1.1 Coprocessor 0 State

Much of the hardware initialization occurs in Coprocessor 0.

- *Status<sub>BEV</sub>* - cleared to 1 on Reset/SoftReset
- *Status<sub>TS</sub>* - cleared to 0 on Reset/SoftReset
- *Status<sub>SR</sub>* - cleared to 0 on Reset, set to 1 on SoftReset
- *Status<sub>NMI</sub>* - cleared to 0 on Reset/SoftReset
- *Status<sub>ERL</sub>* - set to 1 on Reset/SoftReset
- *Status<sub>RP</sub>* - cleared to 0 on Reset/SoftReset
- *Config* fields related to static inputs - set to input value by Reset/SoftReset
- *Config<sub>K0</sub>* - set to 010 (uncached) on Reset/SoftReset
- *DebugDM* - cleared to 0 on Reset/SoftReset (unless EJTAGBOOT option is used to boot into DebugMode, see [Chapter 8, “EJTAG Debug Support in the M14K™ Core”](#) on page 159 for details)

- *Debug<sub>LSNM</sub>* - cleared to 0 on Reset/SoftReset
- *Debug<sub>IBusEP</sub>* - cleared to 0 on Reset/SoftReset
- *Debug<sub>DBusEP</sub>* - cleared to 0 on Reset/SoftReset
- *Debug<sub>IEXI</sub>* - cleared to 0 on Reset/SoftReset
- *Debug<sub>SSI</sub>* - cleared to 0 on Reset/SoftReset

### 6.1.2 Bus State Machines

All pending bus transactions are aborted and the state machines in the SRAM interface unit are reset when a Reset or SoftReset exception is taken.

### 6.1.3 Static Configuration Inputs

All static configuration inputs should only be changed during Reset.

### 6.1.4 Fetch Address

Upon Reset/SoftReset, unless the EJTAGBOOT option is used, the fetch is directed to VA 0xBFC00000 (PA 0x1FC00000). This address is in KSeg1, which is unmapped and uncached.

## 6.2 Software Initialized Processor State

Software is required to initialize the following parts of the device.

### 6.2.1 Register File

The register file powers up in an unknown state with the exception of r0 which is always 0. Initializing the rest of the register file is not required for proper operation in hardware. However, when simulating the operation of the core, unknown values can cause problems. Thus, initializing the register file in the boot code may avoid simulation problems.

### 6.2.2 Coprocessor 0 State

Miscellaneous COP0 states need to be initialized prior to leaving the boot code. There are various exceptions which are blocked by *ERL*=1 or *EXL*=1 and which are not cleared by Reset. These can be cleared to avoid taking spurious exceptions when leaving the boot code.

- *Cause*: WP (Watch Pending), SW0/1 (Software Interrupts) should be cleared.
- *Config*: Typically, the K0, KU and K23 fields should be set to the desired Cache Coherency Algorithm (CCA) value prior to accessing the corresponding memory regions. But in the M14K core, all CCA values are treated identically, so the hardware reset value of these fields need not be modified.
- *Count*: Should be set to a known value if Timer Interrupts are used.

- *Compare*: Should be set to a known value if Timer Interrupts are used. The write to compare will also clear any pending Timer Interrupts (Thus, *Count* should be set before *Compare* to avoid any unexpected interrupts).
- *Status*: Desired state of the device should be set.
- Other COP0 state: Other registers should be written before they are read. Some registers are not explicitly writeable, and are only updated as a by-product of instruction execution or a taken exception. Uninitialized bits should be masked off after reading these registers.



## Power Management of the M14K™ Core

An M14K processor core offers a number of power management features, including low-power design, active power management and power-down modes of operation. The core is a static design that supports a WAIT instruction designed to signal the rest of the device that execution and clocking should be halted, reducing system power consumption during idle periods.

The core provides two mechanisms for system level low-power support discussed in the following sections.

- [Section 7.1 “Register-Controlled Power Management”](#)
- [Section 7.2 “Instruction-Controlled Power Management”](#)

### 7.1 Register-Controlled Power Management

The *RP* bit in the CP0 *Status* register enables a standard software mechanism for placing the system into a low power state. The state of the *RP* bit is available externally via the *SL\_RP* output signal. Three additional pins, *SL\_EXL*, *SL\_ERL*, and *EJ\_DebugM* support the power management function by allowing the user to change the power state if an exception or error occurs while the core is in a low power state.

Setting the *RP* bit of the CP0 *Status* register causes the core to assert the *SL\_RP* signal. The external agent can then decide whether to reduce the clock frequency and place the core into power down mode.

If an interrupt is taken while the device is in power down mode, that interrupt may need to be serviced depending on the needs of the application. The interrupt causes an exception which in turn causes the *EXL* bit to be set. The setting of the *EXL* bit causes the assertion of the *SL\_EXL* signal on the external bus, indicating to the external agent that an interrupt has occurred. At this time the external agent can choose to either speed up the clocks and service the interrupt or let it be serviced at the lower clock speed.

The setting of the *ERL* bit causes the assertion of the *SL\_ERL* signal on the external bus, indicating to the external agent that an error has occurred. At this time the external agent can choose to either speed up the clocks and service the error or let it be serviced at the lower clock speed.

Similarly, the *EJ\_DebugM* signal indicates that the processor is in debug mode. Debug mode is entered when the processor takes a debug exception. If fast handling of this is desired, the external agent can speed up the clocks.

The core provides four power down signals that are part of the system interface. Three of the pins change state as the corresponding bits in the CP0 *Status* register are set or cleared. The fourth pin indicates that the processor is in debug mode:

- The *SL\_RP* signal represents the state of the *RP* bit (27) in the CP0 *Status* register.
- The *SL\_EXL* signal represents the state of the *EXL* bit (1) in the CP0 *Status* register.
- The *SL\_ERL* signal represents the state of the *ERL* bit (2) in the CP0 *Status* register.

- The *EJ\_DebugM* signal indicates that the processor has entered debug mode.

## 7.2 Instruction-Controlled Power Management

The second mechanism for invoking power down mode is through execution of the WAIT instruction. If the bus is idle at the time the WAIT instruction reaches the M stage of the pipeline the internal clocks are suspended and the pipeline is frozen. However, the internal timer and some of the input pins (*SI\_Int[5:0]*, *SI\_NMI*, *SI\_Reset*, *SI\_ColdReset*, and *EJ\_DINT*) continue to run. If the bus is not idle at the time the WAIT instruction reaches the M stage, the pipeline stalls until the bus becomes idle, at which time the clocks are stopped. Once the CPU is in instruction controlled power management mode, any enabled interrupt, NMI, debug interrupt, or reset condition causes the CPU to exit this mode and resume normal operation. While the part is in this low-power mode, the *SI\_SLEEP* signal is asserted to indicate to external agents what the state of the chip is.

## EJTAG Debug Support in the M14K™ Core

The EJTAG debug logic in the M14K processor core provides three optional modules:

1. Hardware breakpoints
2. Test Access Port (TAP) for a dedicated connection to a debug host
3. Tracing of program counter/data address/data value trace to On-chip memory or to a Trace probe

These features are covered in the following sections:

- [Section 8.1 “Debug Control Register”](#)
- [Section 8.2 “Hardware Breakpoints”](#)
- [Section 8.3 “Complex Breakpoint Usage”](#)
- [Section 8.4 “Test Access Port \(TAP\)”](#)
- [Section 8.5 “EJTAG TAP Registers”](#)
- [Section 8.6 “TAP Processor Accesses”](#)
- [Section 8.7 “SecureDebug”](#)
- [Section 8.8 “iFlowtrace™ Mechanism”](#)
- [Section 8.9 “PC/Data Address Sampling”](#)
- [Section 8.10 “Fast Debug Channel”](#)
- [Section 8.11 “cJTAG Interface”](#)

### 8.1 Debug Control Register

The *Debug Control Register (DCR)* register controls and provides information about debug issues, and is always provided with the CPU core. The register is memory-mapped in *drseg* at offset 0x0.

The *DataBrk* and *InstBrk* bits indicate if hardware breakpoints are included in the implementation, and debug software is expected to read hardware breakpoint registers for additional information.

Hardware and software interrupts are maskable for non-debug mode with the *INTE* bit, which works in addition to the other mechanisms for interrupt masking and enabling. NMI is maskable in non-debug mode with the *NMIE* bit, and a pending NMI is indicated through the *NMIP* bit.

The *SRE* bit allows implementation dependent masking of none, some or all sources for soft reset. The soft reset masking may only be applied to a soft reset source if that source can be efficiently masked in the system, thus resulting in no reset at all. If that is not possible, then that soft reset source should not be masked, since a partial soft reset may cause the system to fail or hang. There is no automatic indication of whether the *SRE* is effective, so the user must consult system documentation.

The *PE* bit reflects the *ProbEn* bit from the *EJTAG Control* register (*ECR*), whereby the probe can indicate to the debug software running on the CPU if the probe expects to service dmseg accesses. The reset value in the table below takes effect on both hard and soft resets.

**Figure 8.1 DCR Register Format**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	ENM	0	PCIM	PCno ASID	DASQ	DASe	DAS	0	FDC Impl	Data Brk	Inst Brk				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IVM	DVM	0	RD Vec	CBT	PCS	PCR	PCSe	IntE	NMIE	NMI pend	SRstE	Prob En			

**Table 8.1 DCR Register Field Descriptions**

Fields		Description	Read / Write	Reset State						
Name	Bits									
ENM	29	Endianess in which the processor is running in kernel and Debug Mode: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Little endian</td></tr><tr><td>1</td><td>Big endian</td></tr></table>	Encoding	Meaning	0	Little endian	1	Big endian	R	Preset
Encoding	Meaning									
0	Little endian									
1	Big endian									
PCIM	26	Configure PC Sampling to capture all executed addresses or only those that miss the instruction cache This feature is not supported and this bit will read as 0. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>All PCs captured</td></tr><tr><td>1</td><td>Capture only PCs that miss the cache.</td></tr></table>	Encoding	Meaning	0	All PCs captured	1	Capture only PCs that miss the cache.	R	0
Encoding	Meaning									
0	All PCs captured									
1	Capture only PCs that miss the cache.									
PCnoASID	25	Controls whether the PCSAMPLE scan chain includes or omits the ASID field ASID is always included so this bit will read as 0. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>ASID included in PCSAMPLE scan</td></tr><tr><td>1</td><td>ASID omitted from PCSAMPLE scan</td></tr></table>	Encoding	Meaning	0	ASID included in PCSAMPLE scan	1	ASID omitted from PCSAMPLE scan	R	0
Encoding	Meaning									
0	ASID included in PCSAMPLE scan									
1	ASID omitted from PCSAMPLE scan									



Table 8.1 DCR Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State						
Name	Bits									
DASQ	24	Qualifies Data Address Sampling using a data break-point. Data address sampling is not supported so this bit will read as 0 <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>All data addresses are sampled</td></tr><tr><td>1</td><td>Sample matches of data breakpoint 0</td></tr></table>	Encoding	Meaning	0	All data addresses are sampled	1	Sample matches of data breakpoint 0	R	0
Encoding	Meaning									
0	All data addresses are sampled									
1	Sample matches of data breakpoint 0									
DASe	23	Enables Data Address Sampling Data address sampling is not supported so this bit will read as 0 <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Data Address sampling disabled.</td></tr><tr><td>1</td><td>Data Address sampling enabled.</td></tr></table>	Encoding	Meaning	0	Data Address sampling disabled.	1	Data Address sampling enabled.	R	0
Encoding	Meaning									
0	Data Address sampling disabled.									
1	Data Address sampling enabled.									
DAS	22	Indicates if the Data Address Sampling feature is implemented. Data address sampling is not supported so this bit will read as 0. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No DA Sampling implemented</td></tr><tr><td>1</td><td>DA Sampling implemented</td></tr></table>	Encoding	Meaning	0	No DA Sampling implemented	1	DA Sampling implemented	R	0
Encoding	Meaning									
0	No DA Sampling implemented									
1	DA Sampling implemented									
FDCImpl	18	Indicates if the fast debug channel is implemented <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No fast debug channel implemented</td></tr><tr><td>1</td><td>Fast debug channel implemented</td></tr></table>	Encoding	Meaning	0	No fast debug channel implemented	1	Fast debug channel implemented	R	1
Encoding	Meaning									
0	No fast debug channel implemented									
1	Fast debug channel implemented									
DataBrk	17	Indicates if data hardware breakpoint is implemented: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No data hardware breakpoint implemented</td></tr><tr><td>1</td><td>Data hardware breakpoint implemented</td></tr></table>	Encoding	Meaning	0	No data hardware breakpoint implemented	1	Data hardware breakpoint implemented	R	Preset
Encoding	Meaning									
0	No data hardware breakpoint implemented									
1	Data hardware breakpoint implemented									
InstBrk	16	Indicates if instruction hardware breakpoint is implemented: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No instruction hardware breakpoint implemented</td></tr><tr><td>1</td><td>Instruction hardware breakpoint implemented</td></tr></table>	Encoding	Meaning	0	No instruction hardware breakpoint implemented	1	Instruction hardware breakpoint implemented	R	Preset
Encoding	Meaning									
0	No instruction hardware breakpoint implemented									
1	Instruction hardware breakpoint implemented									

Table 8.1 DCR Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State						
Name	Bits									
IVM	15	Indicates if inverted data value match on data hardware breakpoints is implemented: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No inverted data value match on data hardware breakpoints implemented</td></tr><tr><td>1</td><td>Inverted data value match on data hardware breakpoints implemented</td></tr></table>	Encoding	Meaning	0	No inverted data value match on data hardware breakpoints implemented	1	Inverted data value match on data hardware breakpoints implemented	R	Preset
Encoding	Meaning									
0	No inverted data value match on data hardware breakpoints implemented									
1	Inverted data value match on data hardware breakpoints implemented									
DVM	14	Indicates if a data value store on a data value breakpoint match is implemented: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No data value store on a data value breakpoint match implemented</td></tr><tr><td>1</td><td>Data value store on a data value breakpoint match implemented</td></tr></table>	Encoding	Meaning	0	No data value store on a data value breakpoint match implemented	1	Data value store on a data value breakpoint match implemented	R	Preset
Encoding	Meaning									
0	No data value store on a data value breakpoint match implemented									
1	Data value store on a data value breakpoint match implemented									
RDVec	11	Enables relocation of the debug exception vector. The value in the DebugVectorAddr register is used for EJTAG exceptions when <i>ProbTrap</i> =0,and <i>RDVec</i> =1.	R/W	0						
CBT	10	Indicates if complex breakpoint block is implemented: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No complex breakpoint block implemented</td></tr><tr><td>1</td><td>Complex breakpoint block implemented</td></tr></table>	Encoding	Meaning	0	No complex breakpoint block implemented	1	Complex breakpoint block implemented	R	Preset
Encoding	Meaning									
0	No complex breakpoint block implemented									
1	Complex breakpoint block implemented									
PCS	9	Indicates if the PC Sampling feature is implemented. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No PC Sampling implemented</td></tr><tr><td>1</td><td>PC Sampling implemented</td></tr></table>	Encoding	Meaning	0	No PC Sampling implemented	1	PC Sampling implemented	R	1
Encoding	Meaning									
0	No PC Sampling implemented									
1	PC Sampling implemented									
PCR	8:6	PC Sampling rate. Values 0 to 7 map to values 2 <sup>5</sup> to 2 <sup>12</sup> cycles, respectively. That is, a PC sample is written out every 32, 64, 128, 256, 512, 1024, 2048, or 4096 cycles respectively. The external probe or software is allowed to set this value to the desired sample rate.	R/W	7						
PCSe	5	If the PC sampling feature is implemented, then indicates whether PC sampling is initiated or not. That is, a value of 0 indicates that PC sampling is not enabled and when the bit value is 1, then PC sampling is enabled and the counters are operational.	R/W	0						

Table 8.1 DCR Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State						
Name	Bits									
IntE	4	Hardware and software interrupt enable for Non-Debug Mode, in conjunction with other disable mechanisms: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Interrupt disabled</td></tr><tr><td>1</td><td>Interrupt enabled depending on other enabling mechanisms</td></tr></table>	Encoding	Meaning	0	Interrupt disabled	1	Interrupt enabled depending on other enabling mechanisms	R/W	1
Encoding	Meaning									
0	Interrupt disabled									
1	Interrupt enabled depending on other enabling mechanisms									
NMIE	3	Non-Maskable Interrupt (NMI) enable for Non-Debug Mode: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>NMI disabled</td></tr><tr><td>1</td><td>NMI enabled</td></tr></table>	Encoding	Meaning	0	NMI disabled	1	NMI enabled	R/W	1
Encoding	Meaning									
0	NMI disabled									
1	NMI enabled									
NMIpend	2	Indication for pending NMI: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No NMI pending</td></tr><tr><td>1</td><td>NMI pending</td></tr></table>	Encoding	Meaning	0	No NMI pending	1	NMI pending	R	0
Encoding	Meaning									
0	No NMI pending									
1	NMI pending									
SRstE	1	Soft Reset Enable This bit allows the system to mask soft resets. The core does not internally mask soft resets. Rather the state of this bit appears on the <i>EJ_SRstE</i> external output signal, allowing the system to mask soft resets if desired.	R/W	1						
ProbEn	0	Probe Enable. This bit reflects the ProbEn bit in the EJTAG Control register: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No accesses to dmseg allowed</td></tr><tr><td>1</td><td>EJTAG probe services accesses to dmseg</td></tr></table>	Encoding	Meaning	0	No accesses to dmseg allowed	1	EJTAG probe services accesses to dmseg	R	Same value as ProbEn in ECR (see Table 9-4)
Encoding	Meaning									
0	No accesses to dmseg allowed									
1	EJTAG probe services accesses to dmseg									
0	31:30, 28:27, 21:19, 13:12	Must be written as zeros; return zeros on reads.	0	0						

## 8.2 Hardware Breakpoints

Hardware breakpoints provide for the comparison by hardware of executed instructions and data load/store transactions. It is possible to set instruction breakpoints on addresses even in ROM area. Data breakpoints can be set to cause a debug exception on a specific data transaction. Instruction and data hardware breakpoints are alike for many aspects, and are thus described in parallel in the following. The term hardware is not generally added to breakpoint, unless required to distinguish it from a software breakpoint.

There are two types of simple hardware breakpoints implemented in the M14K core: Instruction breakpoints and Data breakpoints. The M14K core may also contain a complex breakpoint unit.

A core may be configured with the following breakpoint options:

- No data or instruction breakpoints, without complex break support
- Two instruction and one data breakpoint, without complex break support
- Four instruction and two data breakpoints, without complex break support
- Six instruction and two data breakpoints, without support for complex breaks
- Six instruction and two data breakpoints, with support for complex breaks
- Eight instruction and four data breakpoints, without support for complex breaks
- Eight instruction and four data breakpoints, with support for complex breaks

Instruction breaks occur on instruction fetch operations, and the break is set on the virtual address on the bus between the CPU and the instruction cache. Finally, a mask can be applied to the virtual address to set breakpoints on a range of instructions.

Instruction breakpoints compare the virtual address of the executed instructions (the value of PC) with the registers for each instruction breakpoint, including masking of address. When an instruction breakpoint matches, a debug exception and/or a trigger is generated. An internal bit in the instruction breakpoint registers is set to indicate that the match occurred.

### 8.2.1 Data Breakpoints

Data breakpoints occur on load/store transactions. Breakpoints are set on virtual address values, similar to the Instruction breakpoint. Data breakpoints can be set on a load, a store, or both. Data breakpoints can also be set based on the value of the load/store operation. Finally, masks can be applied to both the virtual address and the load/store value.

Data breakpoints compare the transaction type (TYPE), which may be load or store, the virtual address of the transaction (ADDR), accessed bytes (BYTELANE) and data value (DATA), with the registers for each data breakpoint including masking or qualification on the transaction properties. When a data breakpoint matches, a debug exception and/or a trigger is generated, and an internal bit in the data breakpoint registers is set to indicate that the match occurred. The match is precise in that the debug exception or trigger occurs on the instruction that caused the breakpoint to match.

### 8.2.2 Complex Breakpoints

The complex breakpoint unit utilizes the instruction and data breakpoint hardware and looks for more specific matching conditions. There are several different types of enabling that allow more exact breakpoint specification. Tuples add an additional condition to data breakpoints of requiring an instruction breakpoint on the same instructions. Pass counters are counters that decrement each time a matching breakpoint condition is taken. Once the counter reaches 0, the break or trigger effect of the breakpoint is enabled. Priming allows a breakpoint to only be enabled once another trigger condition has been detected. Data qualification allows instruction breakpoints to only be enabled once a corresponding load data triggerpoint has matched both address and data. Data qualified breakpoints are also disabled if a load is executed that matches on the address portion of the triggerpoint, but has a mismatching data value. The complex breakpoint features can be combined to create very complex sequences to match on.

In addition to the breakpoint logic, the complex break unit also includes a Stopwatch Timer block. This counter can be used to measure time spent in various sections. It can either be free-running, or it can be set up to start and stop counting based on a trigger from instruction breakpoints.

### 8.2.3 Conditions for Matching Breakpoints

A number of conditions must be fulfilled in order for a breakpoint to match on an executed instruction or a data transaction, and the conditions for matching instruction and data breakpoints are described below. The breakpoints only match for instructions executed in non-debug mode, thus never on instructions executed in debug mode.

The match of an enabled breakpoint can either generate a debug exception or a trigger indication. The *BE* and/or *TE* bits in the *IBCn* or *DBCn* registers are used to enable the breakpoints.

Debug software should not configure breakpoints to compare on an ASID value unless a TLB is present in the implementation.

#### 8.2.3.1 Conditions for Matching Instruction Breakpoints

There are two methods for matching conditions: Equality and Mask or Address Range.

##### **Equality and Mask**

When an instruction breakpoint is enabled, that breakpoint is evaluated for the address of every executed instruction in non-debug mode, including execution of instructions at an address causing an address error on an instruction fetch. The breakpoint is not evaluated on instructions from a speculative fetch or execution, nor for addresses which are unaligned with an executed instruction.

A breakpoint match depends on the virtual address of the executed instruction (PC) which can be masked at bit level. The registers for each instruction breakpoint have the values and mask used in the compare, and the equation that determines the match is shown below in C-like notation.

$$IB\_match = ( \text{<all 1's>} == ( IBMn_{IBM} \mid \sim ( PC \wedge IBAn_{IBA} ) ) )$$

The match indication for instruction breakpoints is always precise, i.e. indicated on the instruction causing the *IB\_match* to be true.

##### **Address Range**

Cores may optionally support the address range triggered instruction breakpoints. When this feature is configured, the following changes are made to the instruction breakpoint registers:

- *IBAn* : represents the upper limit of a address range boundary
- *IBMn* : represents the lower limit of the address range boundary

In addition, the following bits must be supported:

*IBCn[6].hwarts* : a preset value of 1 indicates that the address range triggered instruction breakpoint feature is supported for this particular instruction breakpoint channel. This bit is read-only.

*IBCn[5].excl* : a value of 0 indicates that the breakpoint will match for addresses inclusive (within) the range defined by *IBMn* and *IBAn*. A value of 1 indicates that the breakpoint will match for addresses exclusive (outside) to the range defined by *IBMn* and *IBAn*. This bit is writeable.

*IBCn[4].hwart* : a value of 0 indicates that the breakpoint will match using the “Equality and Mask” equation as found in Section 8.2.3.1 “Conditions for Matching Instruction Breakpoints”. A value of 1 indicates that the breakpoint will match using address ranges using the equation below:

```
IB_match =
( !IBCnTCuse || ( TC == IBCnTC ) ) &&
( ! IBCnASIDuse || ( ASID == IBASIDnASID ) ) &&
( ( (~IBCnhwarts || ~IBCnhwart) &&
  ( ( IBMnIBM | ~ ( PC ^ IBAnIBA ) ) == ~0 ) ||
  ( ( IBCnhwarts && IBCnhwart) &&
    ( (~IBCnexcl && ( IBM <= PC <= IBA ) ) ||
      ( IBCnexcl && ( IBM > PC || PC > IBA )
    )
  )
)
```

Or if microMIPS is supported:

```
IB_range_match =
( !IBCnTCuse || ( TC == IBCnTC ) ) &&
( ! IBCnASIDuse || ( ASID == IBASIDnASID ) ) &&
( ( (~IBCnhwarts || ~IBCnhwart) &&
  ( ( IBMnIBM | ~ ( ( PC[MSB:1] << 1 ) + ISAmode ) ^ IBAnIBA ) ) == ~0 ) ||
  ( ( IBCnhwarts && IBCnhwart) &&
    ( IBMnIBM[0] | ~ ( ISAmode ^ IBAnIBA[0] ) ) == ~0 ) &&
    ( (~IBCnexcl && ( IBM[MSB:1] <= PC[MSB:1] <= IBA[MSB:1] ) ) ||
      ( IBCnexcl && ( IBM[MSB:1] > PC[MSB:1] || PC[MSB:1] > IBA[MSB:1] )
    )
  )
)
```

Also note that addresses that overlap a boundary is considered for both exclusive and inclusive breakpoint matches.

### 8.2.3.2 Conditions for Matching Data Breakpoints

There are two methods for matching conditions, namely 1) by Equality and Mask or 2) by Address Range:

#### **Equality and Mask**

When a data breakpoint is enabled, that breakpoint is evaluated for every data transaction due to a load/store instruction executed in non-debug mode, including load/store for coprocessor, and transactions causing an address error on data access. The breakpoint is not evaluated due to a PREF instruction or other transactions which are not part of explicit load/store transactions in the execution flow, nor for addresses which are not the explicit load/store source or destination address.

A breakpoint match depends on the transaction type (TYPE) as load or store, the address, and optionally the data value of a transaction. The registers for each data breakpoint have the values and mask used in the compare, and the equation that determines the match is shown below in C-like notation.

The overall match equation is the DB\_match.

```
DB_match =
( ( ( TYPE == load ) && ! DBCnNoLB ) ||
  ( ( TYPE == store ) && ! DBCnNoSB ) ) &&
DB_addr_match && ( DB_no_value_compare || DB_value_match )
```

The match on the address part, `DB_addr_match`, depends on the virtual address of the transaction (`ADDR`) and the accessed bytes (`BYTELANE`) where `BYTELANE[0]` is 1 only if the byte at bits [7:0] on the bus is accessed, and `BYTELANE[1]` is 1 only if the byte at bits [15:8] is accessed, etc. The `DB_addr_match` is shown below.

```
DB_addr_match =
    ( <all 1's> == ( DBMnDBM | ~ ( ADDR ^ DBAnDBA ) ) ) &&
    ( <all 0's> != ( ~ BAI & BYTELANE ) )
```

The size of `DBCnBAI` and `BYTELANE` is 4 bits.

Data value compare is included in the match condition for the data breakpoint depending on the bytes (`BYTELANE` as described above) accessed by the transaction, and the contents of breakpoint registers. The `DB_no_value_compare` is shown below.

```
DB_no_value_compare =
    ( <all 1's> == ( DBCnBLM | DBCnBAI | ~ BYTELANE ) )
```

The size of `DBCnBLM`, `DBCnBAI` and `BYTELANE` is 4 bits.

In case a data value compare is required, `DB_no_value_compare` is false, then the data value from the data bus (`DATA`) is compared and masked with the registers for the data breakpoint. The `DBCIVM` bit inverts the sense of the match - if set, the value match term will be high if the data value is not the same as the data in the `DBVn` register. The endianness is not considered in these match equations for value, as the compare uses the data bus value directly, thus debug software is responsible for setup of the breakpoint corresponding with endianness.

```
DB_value_match =
    DBCnIVM ^
    (( (DATA[7:0] == DBVnDBV[7:0]) || ! BYTELANE[0] || DBCnBLM[0] || DBCnBAI[0] ) &&
    ( (DATA[15:8] == DBVnDBV[15:8]) || ! BYTELANE[1] || DBCnBLM[1] || DBCnBAI[1] ) &&
    ( (DATA[23:16] == DBVnDBV[23:16]) || ! BYTELANE[2] || DBCnBLM[2] || DBCnBAI[2] ) &&
    ( (DATA[31:24] == DBVnDBV[31:24]) || ! BYTELANE[3] || DBCnBLM[3] || DBCnBAI[3] ) )
```

The match for a data breakpoint is always precise, since the match expression is fully evaluated at the time the load/store instruction is executed. A true `DB_match` can thereby be indicated on the very same instruction causing the `DB_match` to be true.

### Address Range

Cores may optionally support the address range triggered data breakpoints. When this feature is configured, the following changes are made to the data breakpoint registers:

- `DBAn` : represents the upper limit of a address range boundary
- `DBMn` : represents the lower limit of the address range boundary

In addition, the following bits must be supported:

`DBCn[10].hwarts`: a preset value of 1 indicates that the address range triggered data breakpoint feature is supported for this particular data breakpoint channel. This bit is read-only.

`DBCn[9].exc` : a value of 0 indicates that the breakpoint will match for addresses inclusive (within) the range defined by `DBMn` and `DBAn`. A value of 1 indicates that the breakpoint will match for addresses exclusive (outside) to the range defined by `DBMn` and `DBAn`. This bit is writeable.

*DBCn[8].hwart*: a value of 0 indicates that the breakpoint will match using the “Equality and Mask” equation as found in Section 8.2.3.2 “Conditions for Matching Data Breakpoints”. A value of 1 indicates that the breakpoint will match using address ranges using the equation below:

```
DB_match =
(!DBCnTCuse || ( TC == DBCnTC ) ) &&
( ( ( TYPE == load ) && ! DBCnNoLB ) || ( ( TYPE == store ) && ! DBCnNoSB ) ) &&
DB_addr_range_match && ( DB_no_value_compare || DB_value_match )

DB_addr_range_match =
( ! DBCnASIDuse || ( ASID == DBASIDnASID ) ) &&
( ((~DBCnhwarts || ~DBCnhwart) &&
  (( DBMnDBM | ~ ( ADDR ^ DBAnDBA ) ) == ~0 ) ||
  (( DBCnhwarts && DBCnhwart) &&
    ((~DBCnexcl && (DBMn <= ADDR <= DBAn)) ||
    ( DBCnexcl && (DBMn > ADDR || ADDR > DBAn)
  )
)
```

When address range triggered data breakpoints is enabled, *DBCn.BLM[3:0]* must be set to 4'b1111 because value matching is not supported with this feature. Addresses that overlap a boundary is considered for both exclusive and inclusive breakpoint matches.

## 8.2.4 Debug Exceptions from Breakpoints

Instruction and data breakpoints may be set up to generate a debug exception when the match condition is true, as described below.

### 8.2.4.1 Debug Exception by Instruction Breakpoint

If the breakpoint is enabled by *BE* bit in the *IBCn* register, then a debug instruction break exception occurs if the *IB\_match* equation is true. The corresponding *BS[n]* bit in the *IBS* register is set when the breakpoint generates the debug exception.

The debug instruction break exception is always precise, so the *DEPC* register and *DBD* bit in the *Debug* register point to the instruction that caused the *IB\_match* equation to be true.

The instruction receiving the debug exception does not update any registers due to the instruction, nor does any load or store by that instruction occur. Thus a debug exception from a data breakpoint can not occur for instructions receiving a debug instruction break exception.

The debug handler usually returns to the instruction causing the debug instruction break exception, whereby the instruction is executed. Debug software is responsible for disabling the breakpoint when returning to the instruction, otherwise the debug instruction break exception reoccurs.

### 8.2.4.2 Debug Exception by Data Breakpoint

If the breakpoint is enabled by *BE* bit in the *DBCn* register, then a debug exception occurs when the *DB\_match* condition is true. The corresponding *BS[n]* bit in the *DBS* register is set when the breakpoint generates the debug exception.

A debug data break exception occurs when a data breakpoint indicates a match. In this case the *DEPC* register and *DBD* bit in the *Debug* register points to the instruction that caused the *DB\_match* equation to be true.



The instruction causing the debug data break exception does not update any registers due to the instruction, and the following applies to the load or store transaction causing the debug exception:

- A store transaction is not allowed to complete the store to the memory system.
- A load transaction with no data value compare, i.e. where the `DB_no_value_compare` is true for the match, is not allowed to complete the load.
- A load transaction for a breakpoint with data value compare must occur from the memory system, since the value is required in order to evaluate the breakpoint.

The result of this is that the load or store instruction causing the debug data break exception appears as not executed, with the exception that a load from the memory system does occur for a breakpoint with data value compare, but the register file is not updated by the load.

If both data breakpoints without and with data value compare would match the same transaction and generate a debug exception, then the following rules apply with respect to updating the `BS[n]` bits.

- On both a load and store the `BS[n]` bits are required to be set for all matching breakpoints without a data value compare.
- On a store the `BS[n]` bits are allowed but not required to be set for all matching breakpoints with a data value compare, but either all or none of the `BS[n]` bits must be set for these breakpoints.
- On a load then none of the `BS[n]` bits for breakpoints with data value compare are allowed to be set, since the load is not allowed to occur due to the debug exception from a breakpoint without a data value compare, and a valid data value is therefore not returned.

Any `BS[n]` bit set prior to the match and debug exception are kept set, since `BS[n]` bits are only cleared by debug software.

The debug handler usually returns to the instruction causing the debug data break exception, whereby the instruction is re-executed. This re-execution may result in a repeated load from system memory, since the load may have occurred previously in order to evaluate the breakpoint as described above. I/O devices with side effects on loads may not be re-accessible without changing the system behavior. The Load Data Value register was introduced to capture the value that was read and allow debug software to synthesize the load instruction without re-accessing memory. Debug software is responsible for disabling breakpoints when returning to the instruction, otherwise the debug data break exception will reoccur.

### 8.2.5 Breakpoint Used as Triggerpoint

Both instruction and data hardware breakpoints can be setup by software so that a matching breakpoint does not generate a debug exception, but only an indication through the `BS[n]` bit. The `TE` bit in the `IBCn` or `DBCn` register controls if an instruction or data breakpoint is used as a so-called triggerpoint. The triggerpoints are, like breakpoints, only compared for instructions executed in non-debug mode.

The `BS[n]` bit in the `IBS` or `DBS` register is set when the respective `IB_match` or `DB_match` bit is true.

The triggerpoint feature can be used to start and stop tracing.

## 8.2.6 Instruction Breakpoint Registers

The registers for instruction breakpoints are described below. These registers have implementation information and are used to set up the instruction breakpoints. All registers are in drseg, and the addresses are shown in [Table 8.2](#).

**Table 8.2 Addresses for Instruction Breakpoint Registers**

Offset in drseg	Register Mnemonic	Register Name and Description
0x1000	<i>IBS</i>	Instruction Breakpoint Status
0x1100 + n * 0x100	<i>IBAn</i>	Instruction Breakpoint Address n
0x1108 + n * 0x100	<i>IBMn</i>	Instruction Breakpoint Address Mask n
0x1110 + n * 0x100	<i>IBASIDn</i>	Instruction Breakpoint ASID n
0x1118 + n * 0x100	<i>IBCn</i>	Instruction Breakpoint Control n
0x1120 + n * 0x100	<i>IBCCn</i>	Instruction Breakpoint Complex Control n
0x1128 + n * 0x100	<i>IBPCn</i>	Instruction Breakpoint Pass Counter n
n is breakpoint number in range 0 to 5 (or 3 or 1, depending on the implemented hardware)		

An example of some of the registers; *IBA0* is at offset 0x1100 and *IBC2* is at offset 0x1318.

### 8.2.6.1 Instruction Breakpoint Status (IBS) Register (0x1000)

**Compliance Level:** Implemented only if instruction breakpoints are implemented.

The *Instruction Breakpoint Status (IBS)* register holds implementation and status information about the instruction breakpoints.

**Figure 8.2 IBS Register Format**

31	30	29	28	27	24	23	6	5	0
Res	ASIDsup	Res	BCN	Res				BS	

**Table 8.3 IBS Register Field Descriptions**

Fields		Description	Read/Wr ite	Reset State
Name	Bit(s)			
Res	31	Must be written as zero; returns zero on read.	R	0
ASIDsup	30	Indicates that ASID compare is supported in instruction breakpoints. 0: No ASID compare. 1: ASID compare (IBASIDn register implemented).	R	0
Res	29:28	Must be written as zero; returns zero on read.	R	0
BCN	27:24	Number of instruction breakpoints implemented.	R	0, 2, 4, 6 or 8 <sup>a</sup>
Res	23:8	Must be written as zero; returns zero on read.	R	0

Table 8.3 IBS Register Field Descriptions

Fields		Description	Read/Wr ite	Reset State
Name	Bit(s)			
BS	7:0	Break status for breakpoint n is at BS[n], with n from 0 to 7 <sup>b</sup> . The bit is set to 1 when the condition for the corresponding breakpoint has matched and <i>IBCnTE</i> or <i>IBCnBE</i> are set	R/W	Undefined
[a] Based on actual hardware implemented. [b] In case of fewer than 8 Instruction breakpoints the upper bits become reserved.				

#### 8.2.6.2 Instruction Breakpoint Address n (IBAn) Register (0x1100 + n \* 0x100)

**Compliance Level:** Implemented only for implemented instruction breakpoints.

The *Instruction Breakpoint Address n (IBAn)* register has the address used in the condition for instruction breakpoint n.

Figure 8.3 IBAn Register Format

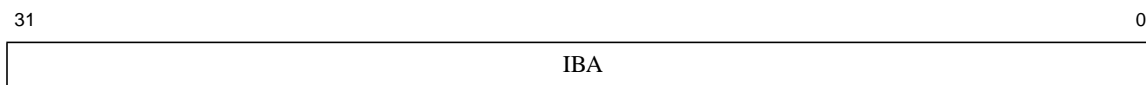


Table 8.4 IBAn Register Field Descriptions

Fields		Description	Read/W rite	Reset State
Name	Bit(s)			
IBA	31:0	Instruction breakpoint address for condition.	R/W	Undefined

#### 8.2.6.3 Instruction Breakpoint Address Mask n (IBMn) Register (0x1108 + n\*0x100)

**Compliance Level:** Implemented only for implemented instruction breakpoints.

The *Instruction Breakpoint Address Mask n (IBMn)* register has the mask for the address compare used in the condition for instruction breakpoint n. A 1 indicates that the corresponding address bit will not be considered in the match. A mask value of all 0's would require an exact address match, while a mask value of all 1's would match on any address.

Figure 8.4 IBMn Register Format



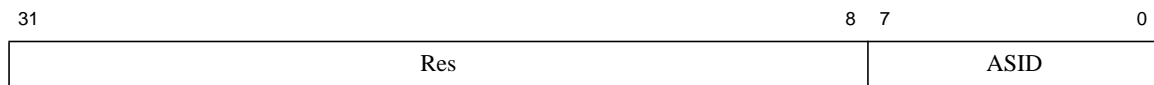
**Table 8.5 IBMn Register Field Descriptions**

Fields		Description	Read/W rite	Reset State						
Name	Bit(s)									
IBM	31:0	Instruction breakpoint address mask for condition:	R/W	Undefined						
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Corresponding address bit not masked.</td></tr><tr><td>1</td><td>Corresponding address bit masked.</td></tr></table>			Encoding	Meaning	0	Corresponding address bit not masked.	1	Corresponding address bit masked.
		Encoding			Meaning					
		0			Corresponding address bit not masked.					
1	Corresponding address bit masked.									

**8.2.6.4 Instruction Breakpoint ASID n (IBASIDn) Register (0x1110 + n\*0x100)**

**Compliance Level:** Implemented only for implemented instruction breakpoints.

For processors with a TLB-based MMU, this register is used to define an ASID value to be used in the match expression. On the M14K processor, this register is reserved and reads as 0.

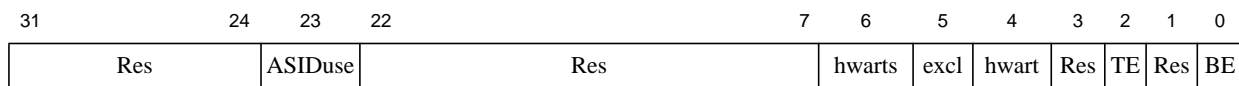
**Figure 8.5 IBASIDn Register Format****Table 8.6 IBASIDn Register Field Descriptions**

Fields		Description	Read/Wr ite	Reset State
Name	Bit(s)			
Res	31:8	Must be written as zero; returns zero on read.	R	0
ASID	7:0	Instruction breakpoint ASID value for a compare.	R	0

**8.2.6.5 Instruction Breakpoint Control n (IBCN) Register (0x1118 + n\*0x100)**

**Compliance Level:** Implemented only for implemented instruction breakpoints.

The *Instruction Breakpoint Control n (IBCN)* register controls the setup of instruction breakpoint *n*.

**Figure 8.6 IBCn Register Format****Table 8.7 IBCn Register Field Descriptions**

Fields		Description	Read/Write	Reset State
Name	Bits			
Res	31:24	Must be written as zero; returns zero on read.	R	0

Table 8.7 IBCn Register Field Descriptions

Fields		Description	Read/Write	Reset State						
Name	Bits									
ASIDuse	23	Use ASID value in compare for instruction breakpoint n: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Don't use ASID value in compare</td></tr><tr><td>1</td><td>Use ASID value in compare</td></tr></table>	Encoding	Meaning	0	Don't use ASID value in compare	1	Use ASID value in compare	R	0
Encoding	Meaning									
0	Don't use ASID value in compare									
1	Use ASID value in compare									
Res	22:7	Must be written as zero; returns zero on read.	R	0						
hwarts	6	A preset value of 1 indicates that the address- range triggered instruction breakpoint feature is supported for this particular instruction breakpoint channel.	R	Preset						
excl	5	A value of 0 indicates that the breakpoint will match for addresses within (inclusive of) the range defined by <i>IBMn</i> and <i>IBAn</i> . A value of 1 indicates that the break-point will match for addresses outside (exclusive to) the range defined by <i>IBMn</i> and <i>IBAn</i> .	R/W	0						
hwart	4	A value of 0 indicates that the breakpoint will match using the “Equality and Mask” equation as found section under 8.2.3.1 “Conditions for Matching Instruction Breakpoints”. A value of 1 indicates that the breakpoint will match using the “Address Range” equation in section 8.2.3.1 “Conditions for Matching Instruction Breakpoints”	R/W	0						
Res	3	Must be written as zero; returns zero on read.	R	0						
TE	2	Use instruction breakpoint n as triggerpoint: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Don't use it as triggerpoint</td></tr><tr><td>1</td><td>Use it as triggerpoint</td></tr></table>	Encoding	Meaning	0	Don't use it as triggerpoint	1	Use it as triggerpoint	R/W	0
Encoding	Meaning									
0	Don't use it as triggerpoint									
1	Use it as triggerpoint									
Res	1	Must be written as zero; returns zero on read.	R	0						
BE	0	Use instruction breakpoint n as breakpoint: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Don't use it as breakpoint</td></tr><tr><td>1</td><td>Use it as breakpoint</td></tr></table>	Encoding	Meaning	0	Don't use it as breakpoint	1	Use it as breakpoint	R/W	0
Encoding	Meaning									
0	Don't use it as breakpoint									
1	Use it as breakpoint									

#### 8.2.6.6 Instruction Breakpoint Complex Control n (IBCCn) Register (0x1120 + n\*0x100)

**Compliance Level:** Implemented only if complex breakpoints are implemented and only for implemented instruction breakpoints.

The Instruction *Breakpoint Complex Control n (IBCCn)* register controls the complex break conditions for instruction breakpoint *n*.

Figure 8.7 IBCCn Register Format

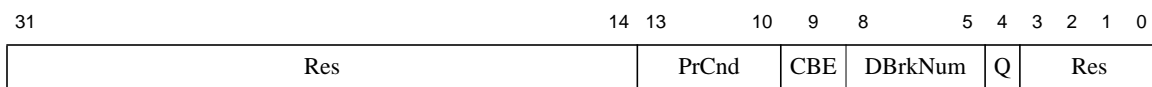


Table 8.8 IBCCn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
Res	31:14, 3:0	Must be written as zero; returns zero on read.	R	0
PrCnd	13:12	Upper bits of priming condition for instruction breakpoint n. The M14K core only supports 4 priming conditions, so the upper 2 bits are read as 0.	R	0
PrCnd	11:10	Priming condition for instruction breakpoint n. 00 - Bypass, no priming needed Other - vary depending on the break number, refer to <a href="#">Table 8.10</a> for mapping.	R/W	0
CBE	9	Complex Break Enable - enables this breakpoint for use in a complex sequence - as a priming condition for another breakpoint, to start or stop the stopwatch timer, or as part of a tuple breakpoint.	R/W	0
DBrkNum	8:5	Indicates which data breakpoint channel is used to qualify this instruction breakpoint.	R	6I/2D Complex Breakpoint Configuration: IBCC0..2 - 0 IBCC3..6 - 1  8I/4D Complex Breakpoint Configuration: IBCC0..1 - 0 IBCC2..3 - 1 IBCC4..5 - 2 IBCC6..7 - 3
Q	4	Qualify this breakpoint based on the data breakpoint indicated in DBrkNum. 0 - Not dependent on qualification 1 - Breakpoint must be qualified to be taken	R/W	0

### 8.2.6.7 Instruction Breakpoint Pass Counter n (IBPCn) Register (0x1128 + n\*0x100)

**Compliance Level:** Implemented only if complex breakpoints are implemented and only for implemented instruction breakpoints.

The Instruction *Breakpoint Pass Counter n (IBPCn)* register controls the pass counter associated with instruction breakpoint n.

If complex breakpoints are implemented, there will be an 8b pass counter for each of the instruction breakpoints on the M14K core.

Figure 8.8 IBPCn Register Format

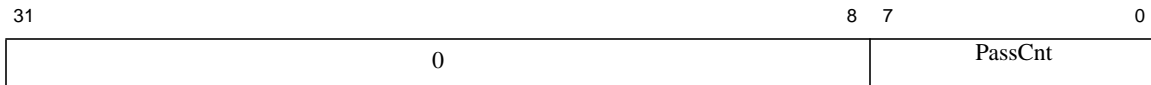


Table 8.9 IBPCn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:8	Ignored on write, returns zero on read.	R	0
PassCnt	7:0	Prevents a break/trigger action until the matching conditions on breakpoint n have been seen this number of times. Each time the matching condition is seen, this value will be decremented by 1. Once the value reaches 0, subsequent matches will cause a break or trigger as requested and the counter will stay at 0. The break or trigger action is imprecise if the PassCnt register was last written to a non-zero value. It will remain imprecise until this register is written to 0 by software. The instruction pass counter should not be set on instruction breakpoints that are being used as part of a tuple breakpoint.	R/W	0

## 8.2.7 Data Breakpoint Registers

The registers for data breakpoints are described below. These registers have implementation information and are used to setup the data breakpoints. All registers are in drseg, and the addresses are shown in [Table 8.10](#).

Table 8.10 Addresses for Data Breakpoint Registers

Offset in drseg	Register Mnemonic	Register Name and Description
0x2000	<i>DBS</i>	Data Breakpoint Status
0x2100 + 0x100 * n	<i>DBAn</i>	Data Breakpoint Address n
0x2108 + 0x100 * n	<i>DBMn</i>	Data Breakpoint Address Mask n
0x2110 + 0x100 * n	<i>DBASIDn</i>	Data Breakpoint ASID n
0x2118 + 0x100 * n	<i>DBCn</i>	Data Breakpoint Control n
0x2120 + 0x100 * n	<i>DBVn</i>	Data Breakpoint Value n
0x2128 + 0x100 * n	<i>DBCCn</i>	Data Breakpoint Complex Control n
0x2130 + 0x100 * n	<i>DBPCn</i>	Data Breakpoint Pass Counter n
0x2ff0	<i>DVM</i>	Data Value Match Register
n is breakpoint number as 0, 1, 2 or 3 (or just 0, depending on the implemented hardware)		

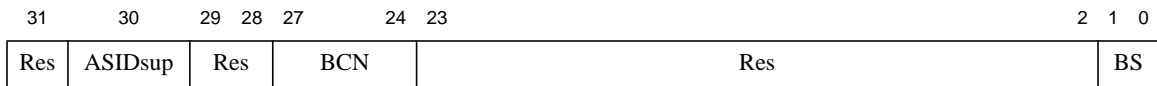
An example of some of the registers; *DBM0* is at offset 0x2108 and *DBV1* is at offset 0x2220.

### 8.2.7.1 Data Breakpoint Status (DBS) Register (0x2000)

**Compliance Level:** Implemented if data breakpoints are implemented.

The *Data Breakpoint Status (DBS)* register holds implementation and status information about the data breakpoints.

**Figure 8.9 DBS Register Format**



**Table 8.11 DBS Register Field Descriptions**

Fields		Description	Read/Wr ite	Reset State
Name	Bit(s)			
Res	31	Must be written as zero; returns zero on read.	R	0
ASID	30	Indicates that ASID compares are supported in data breakpoints. 0: Not supported 1: Supported	R	0
Res	29:28	Must be written as zero; returns zero on read.	R	0
BCN	27:24	Number of data breakpoints implemented.	R	4, 2, 1 or 0 <sup>a</sup>
Res	23:4	Must be written as zero; returns zero on read.	R	0
BS	3:0	Break status for breakpoint n is at BS[n], with n from 0 to 1 <sup>b</sup> . The bit is set to 1 when the condition for the corresponding breakpoint has matched.	R/W0	Undefined

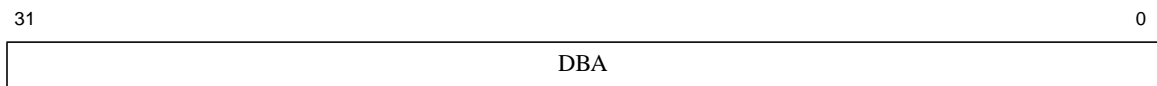
[a] Based on actual hardware implemented.  
[b] In case of only 1 data breakpoint bit 1 become reserved.

### 8.2.7.2 Data Breakpoint Address n (DBAn) Register (0x2100 + 0x100 \* n)

**Compliance Level:** Implemented only for implemented data breakpoints.

The *Data Breakpoint Address n (DBAn)* register has the address used in the condition for data breakpoint *n*.

**Figure 8.10 DBAn Register Format**



**Table 8.12 DBAn Register Field Descriptions**

Fields		Description	Read/W rite	Reset State
Name	Bit(s)			
DBA	31:0	Data breakpoint address for condition.	R/W	Undefined



### 8.2.7.3 Data Breakpoint Address Mask $n$ (DBM $n$ ) Register (0x2108 + 0x100 \* $n$ )

**Compliance Level:** Implemented only for implemented data breakpoints.

The *Data Breakpoint Address Mask  $n$  (DBM $n$ )* register has the mask for the address compare used in the condition for data breakpoint  $n$ . A 1 indicates that the corresponding address bit will not be considered in the match. A mask value of all 0's would require an exact address match, while a mask value of all 1's would match on any address.

**Figure 8.11 DBM $n$  Register Format**



**Table 8.13 DBM $n$  Register Field Descriptions**

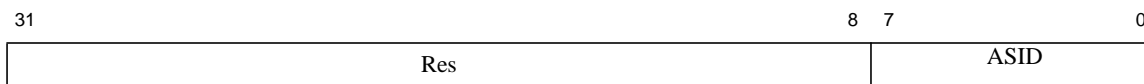
Fields		Description	Read/W rite	Reset State
Name	Bit(s)			
DBM	31:0	Data breakpoint address mask for condition: 0: Corresponding address bit not masked 1: Corresponding address bit masked	R/W	Undefined

### 8.2.7.4 Data Breakpoint ASID $n$ (DBASID $n$ ) Register (0x2110 + 0x100 \* $n$ )

**Compliance Level:** Implemented only for implemented data breakpoints.

For processors with a TLB-based MMU, this register is used to define an ASID value to be used in the match expression. On the M14K processor, this register is reserved and reads as 0.

**Figure 8.12 DBASID $n$  Register Format**



**Table 8.14 DBASID $n$  Register Field Descriptions**

Fields		Description	Read/Wr ite	Reset State
Name	Bit(s)			
Res	31:8	Must be written as zero; returns zero on read.	R	0
ASID	7:0	Data breakpoint ASID value for compares.	R	0

### 8.2.7.5 Data Breakpoint Control $n$ (DBC $n$ ) Register (0x2118 + 0x100 \* $n$ )

**Compliance Level:** Implemented only for implemented data breakpoints.

The *Data Breakpoint Control  $n$  (DBC $n$ )* register controls the setup of data breakpoint  $n$ .

Figure 8.13 DBCn Register Format

31	24	23	22	18	17	14	13	12	11	8	7	4	3	2	1	0
Re	ASIDuse	Res	BAI	NoSB	NoLB	Res	BLM	Res	TE	IVM	BE					

Table 8.15 DBCn Register Field Descriptions

Fields		Description	Read/Write	Reset State						
Name	Bits									
Res	31:24	Must be written as zero; returns zero on reads.	R	0						
ASIDuse	23	Use ASID value in compare for data breakpoint n: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Don't use ASID value in compare</td></tr><tr><td>1</td><td>Use ASID value in compare</td></tr></table>	Encoding	Meaning	0	Don't use ASID value in compare	1	Use ASID value in compare	R	0
Encoding	Meaning									
0	Don't use ASID value in compare									
1	Use ASID value in compare									
Res	22:18	Must be written as zero; returns zero on reads.	R	0						
BAI	17:14	Byte access ignore controls ignore of access to a specific byte. <i>BAI[0]</i> ignores access to byte at bits [7:0] of the data bus, <i>BAI[1]</i> ignores access to byte at bits [15:8], etc. <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Condition depends on access to corresponding byte</td></tr><tr><td>1</td><td>Access for corresponding byte is ignored</td></tr></table>	Encoding	Meaning	0	Condition depends on access to corresponding byte	1	Access for corresponding byte is ignored	R/W	Undefined
Encoding	Meaning									
0	Condition depends on access to corresponding byte									
1	Access for corresponding byte is ignored									
NoSB	13	Controls if condition for data breakpoint is not fulfilled on a store transaction: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Condition may be fulfilled on store transaction</td></tr><tr><td>1</td><td>Condition is never fulfilled on store transaction</td></tr></table>	Encoding	Meaning	0	Condition may be fulfilled on store transaction	1	Condition is never fulfilled on store transaction	R/W	Undefined
Encoding	Meaning									
0	Condition may be fulfilled on store transaction									
1	Condition is never fulfilled on store transaction									
NoLB	12	Controls if condition for data breakpoint is not fulfilled on a load transaction: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Condition may be fulfilled on load transaction</td></tr><tr><td>1</td><td>Condition is never fulfilled on load transaction</td></tr></table>	Encoding	Meaning	0	Condition may be fulfilled on load transaction	1	Condition is never fulfilled on load transaction	R/W	Undefined
Encoding	Meaning									
0	Condition may be fulfilled on load transaction									
1	Condition is never fulfilled on load transaction									
Res	11:8	Must be written as zero; returns zero on reads.	R	0						
BLM	7:4	Byte lane mask for value compare on data breakpoint. <i>BLM[0]</i> masks byte at bits [7:0] of the data bus, <i>BLM[1]</i> masks byte at bits [15:8], etc.: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Compare corresponding byte lane</td></tr><tr><td>1</td><td>Mask corresponding byte lane</td></tr></table>	Encoding	Meaning	0	Compare corresponding byte lane	1	Mask corresponding byte lane	R/W	Undefined
Encoding	Meaning									
0	Compare corresponding byte lane									
1	Mask corresponding byte lane									

Table 8.15 DBCn Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State						
Name	Bits									
Res	3	Must be written as zero; returns zero on reads.	R	0						
TE	2	Use data breakpoint n as triggerpoint:	R/W	0						
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Don't use it as triggerpoint</td></tr><tr><td>1</td><td>Use it as triggerpoint</td></tr></table>			Encoding	Meaning	0	Don't use it as triggerpoint	1	Use it as triggerpoint
		Encoding			Meaning					
		0			Don't use it as triggerpoint					
1	Use it as triggerpoint									
IVM	1	Invert Value Match. When set, the data value compare will be inverted. i.e., a break or trigger will be taken if the value does not match the specified value	R/W	0						
BE	0	Use data breakpoint n as breakpoint:	R/W	0						
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Don't use it as breakpoint</td></tr><tr><td>1</td><td>Use it as breakpoint</td></tr></table>			Encoding	Meaning	0	Don't use it as breakpoint	1	Use it as breakpoint
		Encoding			Meaning					
		0			Don't use it as breakpoint					
1	Use it as breakpoint									

#### 8.2.7.6 Data Breakpoint Value n (DBVn) Register (0x2120 + 0x100 \* n)

**Compliance Level:** Implemented only for implemented data breakpoints.

The *Data Breakpoint Value n (DBVn)* register has the value used in the condition for data breakpoint n.

Figure 8.14 DBVn Register Format

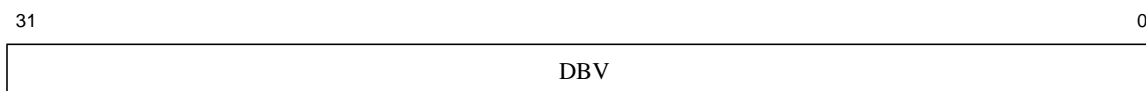


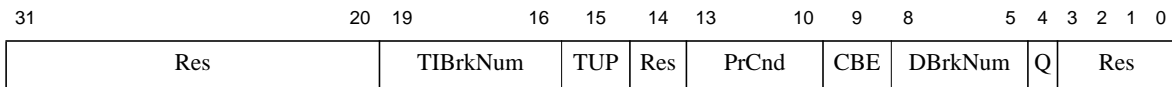
Table 8.16 DBVn Register Field Descriptions

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
DBV	31:0	Data breakpoint value for condition.	R/W	Undefined

#### 8.2.7.7 Data Breakpoint Complex Control n (DBCCn) Register (0x2128 + n\*0x100)

**Compliance Level:** Implemented only if complex breakpoints are implemented and only for implemented data breakpoints.

The Data Breakpoint Complex Control n (*DBCCn*) register controls the complex break conditions for data breakpoint n.

**Figure 8.15 DBCCn Register Format****Table 8.17 DBCCn Register Field Descriptions**

Fields		Description	Read/Write	Reset State
Name	Bits			
Res	31:20, 14, 3:0	Must be written as zero; returns zero on read.	R	0
TIBrkNum	19:16	Tuple Instruction Break Number. Indicates which instruction breakpoint will be paired with this data breakpoint to form a tuple breakpoint	R	6I/2D Complex Breakpoint Configuration: DBCC0 - 0 DBCC1 - 3  8I/4D Complex Breakpoint Configuration: DBCC0 - 0 DBCC1 - 2 DBCC2 - 4 DBCC3 - 6
TUP	15	Tuple Enable. Qualify this data breakpoint with a match on the TIBrkNum instruction breakpoint on the same instruction.	R/W	0
PrCnd	13:12	Upper bits of priming condition for D breakpoint n. M14K only supports 4 priming conditions so the upper 2 bits are read only as 0	R	0
PrCnd	11:10	Priming condition for D Breakpoint n. 00 - Bypass, no priming needed Other - vary depending on the break number, refer to <a href="#">Table 8.20</a> for mapping	R/W	0
CBE	9	Complex Break Enable - enables this breakpoint for use as a priming or qualifying condition for another breakpoint.	R/W	0
DQBrkNum	8:5	Indicates which data breakpoint channel is used to qualify this data breakpoint Data qualification of data breakpoints is not supported on an M14K core and this field will read as 0 and cannot be written.	R	0
DQ	4	Qualify this breakpoint based on the data breakpoint indicated in DBrkNum. Data qualification of data breakpoints is not supported on an M14K core and this field will read as 0 and cannot be written.	R	0

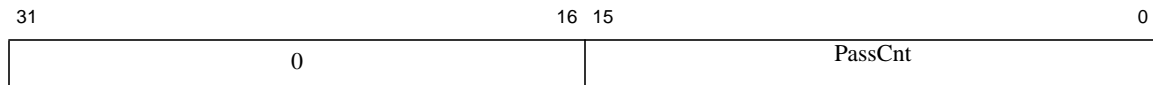
### 8.2.7.8 Data Breakpoint Pass Counter n (DBPCn) Register (0x2130 + n\*0x100)

**Compliance Level:** Implemented only if complex breakpoints are implemented and only for implemented data breakpoints.

The *Data Breakpoint Pass Counter n (DBPCn)* register controls the pass counter associated with data breakpoint n.

If complex breakpoints are implemented, there will be an 16b pass counter for each of the data breakpoints on the M14K core.

**Figure 8.16 DBPCn Register Format**



**Table 8.18 DBPCn Register Field Descriptions**

Fields		Description	Read/Write	Reset State
Name	Bits			
0	31:16	Ignored on write, returns zero on read.	R	0
PassCnt	15:0	Prevents a break/trigger action until the matching conditions on data breakpoint n have been seen this number of times. Each time the matching condition is seen, this value will be decremented by 1. Once the value reaches 0, subsequent matches will cause a break or trigger as requested and the counter will stay at 0. The break or trigger action is imprecise if the PassCnt register was last written to a non-zero value. It will remain imprecise until this register is written to 0 by software.	R/W	0

### 8.2.7.9 Data Value Match (DVM) Register (0x2ff0)

**Compliance Level:** Implemented only if data breakpoints are implemented.

The *Data Value Match (DVM)* register captures the data value of a load that takes a precise data value breakpoint. This allows debug software to synthesize the load instruction without re-executing it in case it is to a system register that has destructive reads.

**Figure 8.17 DVM Register Format**



**Table 8.19 DVM Register Field Descriptions**

Fields		Description	Read/Write	Reset State
Name	Bit(s)			
LDV	31:0	Load data value for the last precise load data value breakpoint taken	R	Undefined

## 8.2.8 Complex Breakpoint Registers

The registers for complex breakpoints are described [Table 8.20](#). These registers have implementation information and are used to setup the data breakpoints. All registers are in drseg.

**Table 8.20 Addresses for Complex Breakpoint Registers**

Offset in drseg	Register Mnemonic	Register Name and Description
$0x1120 + 0x100 * n$	<i>IBCCn</i>	Instruction Breakpoint Complex Control n - described above with instruction breakpoint registers
$0x1128 + 0x100 * n$	<i>IBPCn</i>	Instruction Breakpoint Pass Counter n - described above with instruction breakpoint registers
$0x2128 + 0x100 * n$	<i>DBCCn</i>	Data Breakpoint Complex Control n - described above with data breakpoint registers
$0x2130 + 0x100 * n$	<i>DBPCn</i>	Data Breakpoint Pass Counter n - described above with data breakpoint registers
0x8000	<i>CBTControl</i>	Complex Break and Triggerpoint Control - indicates which of the complex breakpoint features are implemented
$0x8300 + 0x20 * n$	<i>PrCndAln</i>	Prime Condition Register A for Instruction breakpoint n
$0x84e0 + 0x20 * n$	<i>PrCndADn</i>	Prime Condition Register A for Data breakpoint n
0x8900	<i>STCtl</i>	Stopwatch Timer Control
0x8908	<i>STCnt</i>	Stopwatch Timer Count
<i>n</i> is breakpoint number from 0 to 7 (range dependent on implemented hardware)		

### 8.2.8.1 Complex Break and Trigger Control (CBTC) Register (0x8000)

**Compliance Level:** Implemented only if complex breakpoints are implemented.

The *CBTC* register contains configuration bits that indicate which features of complex break are implemented as well as a control bit for the stopwatch timer. On an M14K core, if complex break is implemented, all of the separate features will be present.

**Figure 8.18 CBTC Register Format**

31		9	8	7	5	4	3	2	1	0
Res			STMode	Res	STP	PP	DQP	TP	PCP	

**Table 8.21 CBTC Register Field Descriptions**

Fields		Description	Read/Write	Reset State
Name	Bits			
Res	31:9, 7:5	Reserved	R	0

Table 8.21 CBTC Register Field Descriptions (Continued)

Fields		Description	Read/Write	Reset State
Name	Bits			
STMode	8	Stopwatch Timer Mode: controls whether the stopwatch timer is free-running or controlled by triggerpoints 0 - free-running 1 - started and stopped by instruction triggers	R/W	1
STP	4	Stopwatch Timer Present - indicates whether stopwatch timer is implemented.	R	1
PP	3	Priming Present - indicates whether primed breakpoints are supported	R	1
DQP	2	Data Qualify Present - indicates whether data qualified breakpoints are supported.	R	1
TP	1	Tuple Present - indicates whether any tuple breakpoints are implemented	R	1
PCP	0	Pass Counters Present - indicates whether any break-points have pass counters associated with them	R	1

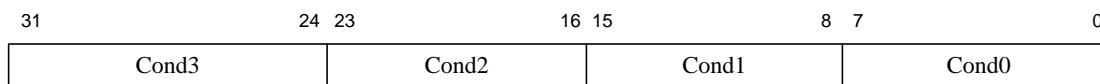
### 8.2.8.2 Priming Condition A (PrCndA/Dn) Registers

**Compliance Level:** Implemented if complex breakpoints are implemented.

The *Prime Condition* registers hold implementation specific information about which triggerpoints are used for the priming conditions for each breakpoint register. On an M14K core, these connections are predetermined and these registers are read-only.

The architecture allows for up to 16 priming conditions to be specified and there can be up to 4 priming condition registers per breakpoint (A/B/C/D). An M14K core only allows for 4 priming conditions and thus only implements the PrCndA registers. The general description is shown in Table 8.22. The actual priming conditions for each of the breakpoints are shown in Table 8.23.

Figure 8.19 PrCndA Register Format



**Table 8.22 PrCndA Register Field Descriptions**

Fields		Description	Read/Wr ite	Reset State
Name	Bit(s)			
CondN	31:24 23:16 15:8 7:0	Specifies which triggerpoint is connected to priming condition 3, 2, 1, or 0 <sup>a</sup> for the current breakpoint.	R	Preset
	31:30 23:22 15:14 7:6	Reserved	R	0
	29:28 21:20 13:12 5:4	Trigger type 00 - Special/Bypass 01 - Instruction 10 - Data 11 - Reserved	R	Preset
	27:24 19:16 11:8 3:0	Break Number, 0-14	R	Preset
[a] Condition 0 is always Bypass and will read as 8'b0				

**Table 8.23 Priming Conditions and Register Values for 6I/2D Configuration**

Break	Cond0	Cond1	Cond2	Cond3	PrCndA Value	drseg offset
Inst0	Bypass	Data0	Inst1	Inst2	0x1211_2000	0x8300
Inst1	Bypass	Data0	Inst0	Inst2	0x1210_2000	0x8320
Inst2	Bypass	Data0	Inst0	Inst1	0x1110_2000	0x8340
Inst3	Bypass	Data1	Inst4	Inst5	0x1514_2100	0x8360
Inst4	Bypass	Data1	Inst3	Inst5	0x1513_2100	0x8380
Inst5	Bypass	Data1	Inst3	Inst4	0x1413_2100	0x83a0
Data0	Bypass	Inst0	Inst1	Inst2	0x1211_1000	0x84e0
Data1	Bypass	Inst3	Inst4	Inst5	0x1514_1300	0x8500

**Table 8.24 Priming Conditions and Register Values for 8I/4D Configuration**

Break	Cond0	Cond1	Cond2	Cond3	PrCndA Value	drseg offset
Inst0	Bypass	Data0	Inst1	Inst2	0x1211_2000	0x8300
Inst1	Bypass	Data0	Inst0	Inst2	0x1210_2000	0x8320
Inst2	Bypass	Data1	Inst3	Inst4	0x1413_2100	0x8340
Inst3	Bypass	Data1	Inst2	Inst4	0x1412_2100	0x8360
Inst4	Bypass	Data2	Inst5	Inst6	0x1615_2200	0x8380



Break	Cond0	Cond1	Cond2	Cond3	PrCndA Value	drseg offset
Inst5	Bypass	Data2	Inst4	Inst6	0x1614_2200	0x83a0
Inst6	Bypass	Data3	Inst7	Inst0	0x1017_2300	0x83c0
Inst7	Bypass	Data3	Inst6	Inst0	0x1016_2300	0x83e0
Data0	Bypass	Inst0	Inst1	Data1	0x2111_1000	0x84e0
Data1	Bypass	Inst2	Inst3	Data2	0x2213_1200	0x8500
Data2	Bypass	Inst4	Inst5	Data3	0x2315_1400	0x8520
Data3	Bypass	Inst6	Inst7	Data0	0x2017_1600	0x8540

### 8.2.8.3 Stopwatch Timer Control (STCtl) Register (0x8900)

**Compliance Level:** Implemented if stopwatch timer is implemented.

The *Stopwatch Timer Control (STCtl)* register gives configuration information about how the stopwatch timer register is controlled. On an M14K core, the break channels that control the stopwatch timer are fixed and this register is read-only.

**Figure 8.20 STCtl Register Format**

31	18	17	14	13	10	9	8	5	4	1	0
Res				StopChan1	StartChan1	En1	StopChan0	StartChan0	En0		

**Table 8.25 STCtl Register Field Descriptions**

Fields		Description	Read/Wr ite	Reset State
Name	Bit(s)			
Res	31:18	Must be written as zero; returns zero on read.	R	0
StopChan1	17:14	Indicates the instruction breakpoint channel that will stop the counter if the timer is under pair1 breakpoint control	R	0
StartChan1	13:10	Indicates the instruction breakpoint channel that will start the counter if the timer is under pair1 breakpoint control	R	0
En1	9	Enables the second pair (pair1) of breakpoint registers to control the timer when under breakpoint control. If the stopwatch timer is configured to be under breakpoint control (by setting <i>CBTControlSTM</i> ) and this bit is set, the breakpoints indicated in the StartChan1 and StopChan1 fields will control the timer.  The M14K core only supports 1 pair of stopwatch control breakpoints so this field is not writable and will read as 0	R	0
StopChan0	8:5	Indicates the instruction breakpoint channel that will stop the counter if the timer is under pair0 breakpoint control	R	0x4

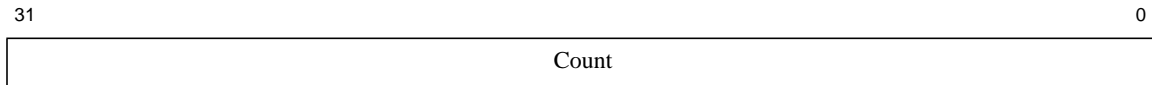
**Table 8.25 STCtI Register Field Descriptions**

Fields		Description	Read/Wr ite	Reset State
Name	Bit(s)			
StartChan0	4:1	Indicates the instruction breakpoint channel that will start the counter if the timer is under pair0 breakpoint control	R	0x1
En0	0	Enables the first pair (pair0) of breakpoint registers to control the timer when under breakpoint control. If the stopwatch timer is configured to be under breakpoint control (by setting <i>CBTControl/STM</i> ) and this bit is set, the breakpoints indicated in the StartChan0 and StopChan0 fields will control the timer.  The M14K core only supports 1 pair of stopwatch control breakpoints so this field is not writable and will read as 1	R	1

**8.2.8.4 Stopwatch Timer Count (STCnt) Register (0x8908)**

**Compliance Level:** Implemented if stopwatch timer is implemented.

The *Stopwatch Timer Count (STCnt)* register is the count value for the stopwatch timer.

**Figure 8.21 STCnt Register Format****Table 8.26 STCtI Register Field Descriptions**

Fields		Description	Read/Wr ite	Reset State
Name	Bit(s)			
Count	31:0	Current counter value	R/W	0

**8.3 Complex Breakpoint Usage****8.3.1 Checking for Presence of Complex Break Support**

Software should verify that the complex breakpoint hardware is implemented prior to attempting to use it. The full sequence of steps is shown below for general use. Spots where the M14K core has restricted behavior are noted.

1. Read the *Config1EP* bit to check for the presence of EJTAG logic. EJTAG logic is always present on an M14K core.

2. Read the *DebugNoDCR* bit to check for the presence of the *Debug Control Register (DCR)*. The *DCR* will always be implemented on an M14K core.
3. Read the *DCRCBT* bit to check for the presence of any complex break and trigger features
4. Read the *CBTControl* register to check for the presence of each individual feature. If an M14K core implements any complex break and trigger features, it will implement all of them
5. If Pass Counters are implemented, they may not be implemented for all break channels and may have different counter sizes. To determine the size and presence of each pass counter, software can write -1 to each of the *IBPCn* and *DBPCn* registers and read it back. If an M14K core implements pass counters, it will implement an 8b counter for each instruction breakpoint and a 16b counter for each data breakpoint.
6. If tuples are implemented, they may only be supported on a subset of the data breakpoint channels. This can be checked by seeing if the *DBBCnTUP* bit can be set to 1. Additionally, some cores may support dynamically changing which instruction breakpoint is associated with a given data breakpoint. This can be checked by attempting to write the *DBCCnTIBrkNum* field. If an M14K core implements tuple support, it will support it for all data breakpoint channels and the instruction breakpoint association will be fixed.
7. If Priming Conditions are supported, a core may only support a subset of the possible priming condition values. This can be checked by writing to the *xBCCnPrCnd* field. If only 1 or 2 bits can be written, the available priming conditions will be described in the *PrCndA* registers. If 3 bits are writable, *PrCndA* and *PrCndB* will describe the conditions, and if all 4 bits are writable, the *PrCndA*, *PrCndB*, *PrCndC*, and *PrCndD* registers will all exist. Some cores may also support changing the priming conditions and this can be checked by attempting to write to the *PrCnd* registers. If an M14K core supports priming conditions, it will support 4 statically mapped priming conditions per breakpoint which will be described in the *PrCndA* registers.
8. If support for qualified breakpoints is indicated, it may only be supported for some of the breakpoints. Additionally, the data breakpoint used for the qualification may be configurable. Software can check this by writing to the *xBCCnDQ* and *xBCCnDQBrkNum* fields. If an M14K core support qualified breakpoints, it will only support it on instruction breakpoints and the data break used for qualification will be fixed for each instruction breakpoint.
9. If the stopwatch timer is implemented, either one or two pairs of instruction breakpoints may be available for controlling it and it may be possible to dynamically select which instruction breakpoints are used. This can be tested by writing to the *STCtI* register.

### 8.3.2 General Complex Break Behavior

There is some general complex break behavior that is common to all complex breakpoints. . This behavior is described below:

- Resets to a disabled state - when the core is reset, the complex break functionality will be disabled and debug software that is not aware of complex break should continue to function normally.
- Complex break state is not updated on exceptional instructions
- Complex breakpoints are evaluated at the end of the pipeline and complex breakpoint exceptions are taken imprecisely on the following instruction.
- There is no hazard between enabling and enabled events. When an instruction causes an enabling event, the following instruction sees the enabled state and reacts accordingly.

### 8.3.3 Usage of Pass Counters

Pass counters specify that the breakpoint conditions must match  $N$  times before the breakpoint action will be enabled.

- Controlled by writing to the per-breakpoint pass counter register
- Resets to 0
- Writing to a non-zero value enables the pass counter. When enabled, each time the breakpoint conditions match, the counter will be decremented by 1. After the counter value reaches 0, the breakpoint action (breakpoint exception, trigger, or complex break enable) will occur on any subsequent matches and the counter will not decrement further. The action does not occur on the match that causes the 1->0 counter decrement.
- If the breakpoint also has priming conditions and/or data qualified specified, the pass counter will only decrement when the priming and/or qualified conditions have been met
- If a data breakpoint is configured to be a tuple breakpoint, the data pass counter will only decrement on instructions where both the instruction and data break conditions match. The pass counter for the instruction break involved in a tuple should not be enabled if the tuple is enabled.
- Once a pass counter has been enabled, it will be treated as enabled until the pass counter is explicitly written to 0. Namely, breakpoint exceptions will continue to be taken imprecisely until the pass counter is disabled by writing to 0.
- The counter register will be updated as matches are detected. The current count value can be read from the register while operating in debug mode. Note that this behavior is architecturally recommended, but not required.

### 8.3.4 Usage of Tuple Breakpoints

A tuple breakpoint is the logical AND of a data breakpoint and an instruction breakpoint. Tuple breakpoints are specified as a condition on a data breakpoint. If the *DBCCnTUP* bit is set, the data breakpoint will not match unless there the corresponding instruction breakpoint conditions are also met.

- Uses the data breakpoint resources to specify the break action, break status, pass counters, and priming conditions.
- The instruction breakpoint involved in the tuple should be configured as follows:
  - $IBCCn_{CBE} = 1$
  - $IBCCn_{PrCnd} = IBCCn_{DQ} = IBCCn_{TE} = IBCCn_{BE} = IBPCn = 0$

### 8.3.5 Usage of Priming Conditions

Priming conditions provide a way to have one breakpoint enabled by another one. Prior to the priming condition being satisfied, any breakpoint matches are ignored.

- Priming condition resets to bypass which specifies that no priming is required
- 3 other priming conditions are available for each breakpoint. These condition vary from breakpoint to breakpoint (since it makes no sense for a breakpoint to prime itself). The conditions for each of the breakpoints are listed in [Table 8.23](#).

- The priming breakpoint must have  $xBCCn_{TE}$  or  $xBCCn_{CBE}$  set.
- Once the priming condition has been seen, the primed breakpoint will remain primed until its  $xBCCn$  register is written
- The primed state is stored with the breakpoint being primed and not with the breakpoint that is doing the priming.
- Each Prime condition is the comparator output after it has been qualified by its own Prime condition, data qualification, and pass counter. Using this, several stages of priming are possible (e.g. data cycle D followed by instruction A followed by instruction B N times followed by instruction C).

### 8.3.6 Usage of Data Qualified Breakpoints

Each of the instruction breakpoints can be set to be data qualified. In qualified mode, a breakpoint will recognize its conditions only after the specified data breakpoint matches both address and data. If the data breakpoint matches address, but has a mismatch on the data value, the instruction breakpoint will be unqualified and will not match until a subsequent qualifying match.

This feature can be used similarly to the ASID qualification that is available on cores with TLBs. If an RTOS loads a process ID for the current process, that load can be used as the qualifying breakpoint. When a matching process ID is loaded (entering the desired RTOS process), qualified instruction breakpoints will be enabled. When a different process ID is loaded (leaving the desired RTOS process), the qualified instruction breakpoints are disabled. Alternatively, with the InvertValueMatch feature of the data breakpoint, the instruction breakpoints could be enabled on any process ID other than the specified one.

- The qualifying data break must have  $DBCn_{TE}$  or  $DBCn_{CBE}$  set.
- The qualifying data break should have data comparison enabled (via settings of  $DBCn_{BLM}$  and  $DBCn_{BAI}$ )
- The qualifying data break should not have pass counters, priming conditions, or tuples enabled.
- The qualifying data access can be either a load or store, depending on the settings of  $DBCn_{NoSB}$  and  $DBCn_{NoLB}$
- The Qualified/Unqualified state is stored with the instruction breakpoint that is being qualified. Writing its  $IBCCn$  register will disqualify that breakpoint.
- Qualified instruction breakpoint can also have priming conditions and/or pass counters enabled. The pass counter will only decrement when the priming and qualifying conditions have been met. The instruction breakpoint action (break, trigger, or complex enable) will only occur when all priming, qualifying, and pass counter conditions have been met.
- Qualified instruction breakpoint can be used to prime another breakpoint

### 8.3.7 Usage of Stopwatch Timers

The stopwatch timer is a drseg memory mapped count register. It can be configured to be free running or controlled by instruction breakpoints. This could be used to measure the amount of time that is spent in a particular function by starting the counter upon function entry and stopping it upon exit.

- Count value is reset to 0

- Reset state has counter stopped and under breakpoint control so that the counter is not running when the core is not being debugged.
- Bit in *CBTControl* register controls whether the counter is free-running or breakpoint controlled.
- Counter does not count in debug mode
- When breakpoint controlled, the involved instruction breakpoints must have *IBCn<sub>TE</sub>* or *IBCCn<sub>CBE</sub>* set in order to start or stop the timer.

## 8.4 Test Access Port (TAP)

The following main features are supported by the TAP module:

- 5-pin industry standard JTAG Test Access Port (*TCK*, *TMS*, *TDI*, *TDO*, *TRST\_N*) interface which is compatible with IEEE Std. 1149.1.
- Target chip and EJTAG feature identification available through the Test Access Port (TAP) controller.
- The processor can access external memory on the EJTAG Probe serially through the EJTAG pins. This is achieved through Processor Access (PA), and is used to eliminate the use of the system memory for debug routines.
- Support for both ROM based debugger and debugging both through TAP.

### 8.4.1 EJTAG Internal and External Interfaces

The external interface of the EJTAG module consists of the 5 signals defined by the IEEE standard.

**Table 8.27 EJTAG Interface Pins**

Pin	Type	Description
<i>TCK</i>	I	Test Clock Input Input clock used to shift data into or out of the Instruction or data registers. The <i>TCK</i> clock is independent of the processor clock, so the EJTAG probe can drive <i>TCK</i> independently of the processor clock frequency. The core signal for this is called <i>EJ_TCK</i>
<i>TMS</i>	I	Test Mode Select Input The <i>TMS</i> input signal is decoded by the TAP controller to control test operation. <i>TMS</i> is sampled on the rising edge of <i>TCK</i> . The core signal for this is called <i>EJ_TMS</i>
<i>TDI</i>	I	Test Data Input Serial input data ( <i>TDI</i> ) is shifted into the Instruction register or data registers on the rising edge of the <i>TCK</i> clock, depending on the TAP controller state. The core signal for this is called <i>EJ_TDI</i>
<i>TDO</i>	O	Test Data Output Serial output data is shifted from the Instruction or data register to the <i>TDO</i> pin on the falling edge of the <i>TCK</i> clock. When no data is shifted out, the <i>TDO</i> is 3-stated. The core signal for this is called <i>EJ_TDO</i> with output enable controlled by <i>EJ_TDOzstate</i> .

Table 8.27 EJTAG Interface Pins (Continued)

Pin	Type	Description
<i>TRST_N</i>	I	<p>Test Reset Input (Optional pin)</p> <p>The <i>TRST_N</i> pin is an active-low signal for asynchronous reset of the TAP controller and instruction in the TAP module, independent of the processor logic. The processor is not reset by the assertion of <i>TRST_N</i>. The core signal for this is called <i>EJ_TRST_N</i>.</p> <p>This signal is optional, but power-on reset must apply a low pulse on this signal at power-on and then leave it high, in case the signal is not available as a pin on the chip. If available on the chip, then it must be low on the board when the EJTAG debug features are unused by the probe.</p>

### 8.4.2 Test Access Port Operation

The TAP controller is controlled by the Test Clock (*TCK*) and Test Mode Select (*TMS*) inputs. These two inputs determine whether an the Instruction register scan or data register scan is performed. The TAP consists of a small controller, driven by the *TCK* input, which responds to the *TMS* input as shown in the state diagram in Figure 8.22. The TAP uses both clock edges of *TCK*. *TMS* and *TDI* are sampled on the rising edge of *TCK*, while *TDO* changes on the falling edge of *TCK*.

At power-up the TAP is forced into the *Test-Logic-Reset* by low value on *TRST\_N*. The TAP instruction register is thereby reset to IDCODE. No other parts of the EJTAG hardware are reset through the *Test-Logic-Reset* state.

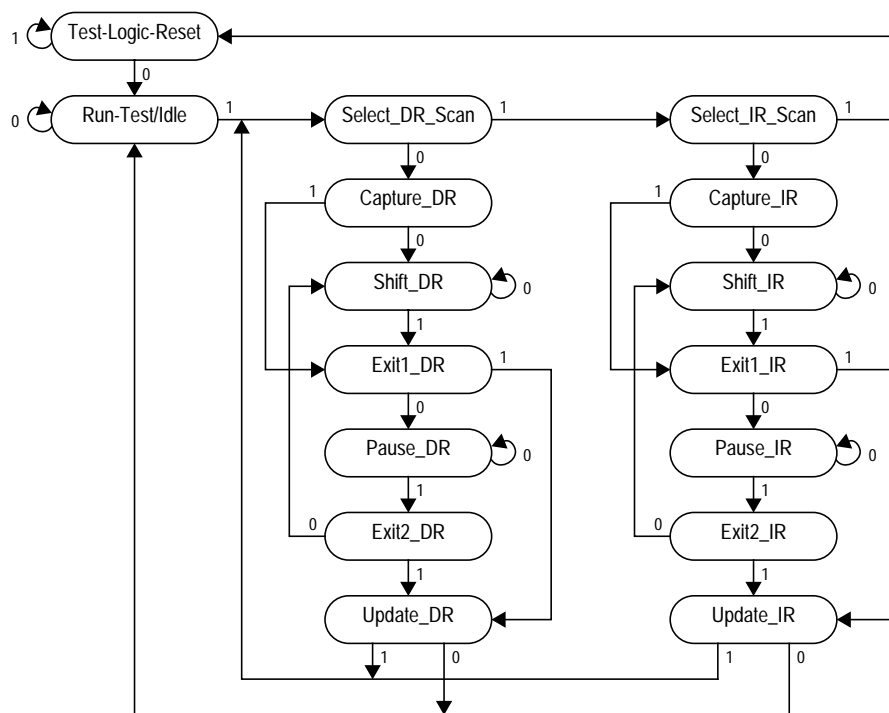
When test access is required, a protocol is applied via the *TMS* and *TCK* inputs, causing the TAP to exit the *Test-Logic-Reset* state and move through the appropriate states. From the *Run-Test/Idle* state, an Instruction register scan or a data register scan can be issued to transition the TAP through the appropriate states shown in Figure 8.22.

The states of the data and instruction register scan blocks are mirror images of each other adding symmetry to the protocol sequences. The first action that occurs when either block is entered is a capture operation. For the data registers, the *Capture-DR* state is used to capture (or parallel load) the data into the selected serial data path. In the Instruction register, the *Capture-IR* state is used to capture status information into the Instruction register.

From the *Capture* states, the TAP transitions to either the *Shift* or *Exit1* states. Normally the *Shift* state follows the *Capture* state so that test data or status information can be shifted out for inspection and new data shifted in. Following the *Shift* state, the TAP either returns to the *Run-Test/Idle* state via the *Exit1* and *Update* states or enters the *Pause* state via *Exit1*. The reason for entering the *Pause* state is to temporarily suspend the shifting of data through either the Data or Instruction Register while a required operation, such as refilling a host memory buffer, is performed. From the *Pause* state shifting can resume by re-entering the *Shift* state via the *Exit2* state or terminate by entering the *Run-Test/Idle* state via the *Exit2* and *Update* states.

Upon entering the data or Instruction register scan blocks, shadow latches in the selected scan path are forced to hold their present state during the Capture and Shift operations. The data being shifted into the selected scan path is not output through the shadow latch until the TAP enters the *Update-DR* or *Update-IR* state. The *Update* state causes the shadow latches to update (or parallel load) with the new data that has been shifted into the selected scan path.

### Figure 8.22 TAP Controller State Diagram



#### 8.4.2.1 Test-Logic-Reset State

In the *Test-Logic-Reset* state the boundary scan test logic is disabled. The test logic enters the *Test-Logic-Reset* state when the *TMS* input is held HIGH for at least five rising edges of *TCK*. The BYPASS instruction is forced into the instruction register output latches during this state. The controller remains in the *Test-Logic-Reset* state as long as *TMS* is HIGH.

#### 8.4.2.2 Run-Test/Idle State

The controller enters the *Run-Test/Idle* state between scan operations. The controller remains in this state as long as *TMS* is held LOW. The instruction register and all test data registers retain their previous state. The instruction cannot change when the TAP controller is in this state.

When **TMS** is sampled HIGH on the rising edge of **TCK**, the controller transitions to the *Select\_DR* state.

#### 8.4.2.3 Select\_DR\_Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, then the controller transitions to the *Capture\_DR* state. A HIGH on *TMS* causes the controller to transition to the *Select\_IR* state. The instruction cannot change while the TAP controller is in this state.

#### 8.4.2.4 Select IR Scan State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If **TMS** is sampled LOW on the rising edge of **TCK**, the controller transitions to the *Capture\_IR* state. A



HIGH on *TMS* causes the controller to transition to the *Test-Reset-Logic* state. The instruction cannot change while the TAP controller is in this state.

#### 8.4.2.5 Capture\_DR State

In this state the boundary scan register captures the value of the register addressed by the Instruction register, and the value is then shifted out in the *Shift\_DR*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift\_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1\_DR* state. The instruction cannot change while the TAP controller is in this state.

#### 8.4.2.6 Shift\_DR State

In this state the test data register connected between *TDI* and *TDO* as a result of the current instruction shifts data one stage toward its serial output on the rising edge of *TCK*. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller remains in the *Shift\_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1\_DR* state. The instruction cannot change while the TAP controller is in this state.

#### 8.4.2.7 Exit1\_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Pause\_DR* state. A HIGH on *TMS* causes the controller to transition to the *Update\_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

#### 8.4.2.8 Pause\_DR State

The *Pause\_DR* state allows the controller to temporarily halt the shifting of data through the test data register in the serial path between *TDI* and *TDO*. All test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW on the rising edge of *TCK*, the controller remains in the *Pause\_DR* state. A HIGH on *TMS* causes the controller to transition to the *Exit2\_DR* state. The instruction cannot change while the TAP controller is in this state.

#### 8.4.2.9 Exit2\_DR State

This is a temporary controller state in which all test data registers selected by the current instruction retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift\_DR* state to allow another serial shift of data. A HIGH on *TMS* causes the controller to transition to the *Update\_DR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

#### 8.4.2.10 Update\_DR State

When the TAP controller is in this state the value shifted in during the *Shift\_DR* state takes effect on the rising edge of the *TCK* for the register indicated by the Instruction register.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Run-Test/Idle* state. A HIGH on *TMS* causes the controller to transition to the *Select\_DR\_Scan* state. The instruction cannot change while the TAP controller is in this state and all shift register stages in the test data registers selected by the current instruction retain their previous state.

#### 8.4.2.11 Capture\_IR State

In this state the shift register contained in the Instruction register loads a fixed pattern (00001<sub>2</sub>) on the rising edge of *TCK*. The data registers selected by the current instruction retain their previous state.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Shift\_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1\_IR* state. The instruction cannot change while the TAP controller is in this state.

#### 8.4.2.12 Shift\_IR State

In this state the instruction register is connected between *TDI* and *TDO* and shifts data one stage toward its serial output on the rising edge of *TCK*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller remains in the *Shift\_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit1\_IR* state.

#### 8.4.2.13 Exit1\_IR State

This is a temporary controller state in which all registers retain their previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Pause\_IR* state. A HIGH on *TMS* causes the controller to transition to the *Update\_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state and the instruction register retains its previous state.

#### 8.4.2.14 Pause\_IR State

The *Pause\_IR* state allows the controller to temporarily halt the shifting of data through the instruction register in the serial path between *TDI* and *TDO*. If *TMS* is sampled LOW at the rising edge of *TCK*, the controller remains in the *Pause\_IR* state. A HIGH on *TMS* causes the controller to transition to the *Exit2\_IR* state. The instruction cannot change while the TAP controller is in this state.

#### 8.4.2.15 Exit2\_IR State

This is a temporary controller state in which the instruction register retains its previous state. If *TMS* is sampled LOW at the rising edge of *TCK*, then the controller transitions to the *Shift\_IR* state to allow another serial shift of data. A HIGH on *TMS* causes the controller to transition to the *Update\_IR* state which terminates the scanning process. The instruction cannot change while the TAP controller is in this state.

#### 8.4.2.16 Update\_IR State

The instruction shifted into the instruction register takes effect on the rising edge of *TCK*.

If *TMS* is sampled LOW at the rising edge of *TCK*, the controller transitions to the *Run-Test/Idle* state. A HIGH on *TMS* causes the controller to transition to the *Select\_DR\_Scan* state.

### 8.4.3 Test Access Port (TAP) Instructions

The TAP Instruction register allows instructions to be serially input into the device when TAP controller is in the *Shift-IR* state. Instructions are decoded and define the serial test data register path that is used to shift data between *TDI* and *TDO* during data register scanning.

The Instruction register is a 5-bit register. In the current EJTAG implementation only some instructions have been decoded; the unused instructions default to the BYPASS instruction.

**Table 8.28 Implemented EJTAG Instructions**

Value	Instruction	Function
0x01	IDCODE	Select Chip Identification data register
0x03	IMPCODE	Select Implementation register
0x08	ADDRESS	Select Address register
0x09	DATA	Select Data register
0x0A	CONTROL	Select EJTAG Control register
0x0B	ALL	Select the Address, Data and EJTAG Control registers
0x0C	EJTAGBOOT	Set EjtagBrk, ProbEn and ProbTrap to 1 as reset value
0x0D	NORMALBOOT	Set EjtagBrk, ProbEn and ProbTrap to 0 as reset value
0x0E	FASTDATA	Selects the Data and Fastdata registers
0x10	TCBCONTROLA	Selects the <i>TCBTCONTROLA</i> register in the Trace Control Block
0x11	TCBCONTROLB	Selects the <i>TCBTCONTROLB</i> register in the Trace Control Block
0x12	TCBDATA	Selects the <i>TCBDATA</i> register in the Trace Control Block
0x14	PCSAMPLE	Selects the PCsample register
0x17	FDC	Selects Fast Debug Channel.
0x1F	BYPASS	Bypass mode

#### 8.4.3.1 BYPASS Instruction

The required BYPASS instruction allows the processor to remain in a functional mode and selects the Bypass register to be connected between *TDI* and *TDO*. The BYPASS instruction allows serial data to be transferred through the processor from *TDI* to *TDO* without affecting its operation. The bit code of this instruction is defined to be all ones by the IEEE 1149.1 standard. Any unused instruction is defaulted to the BYPASS instruction.

#### 8.4.3.2 IDCODE Instruction

The IDCODE instruction allows the processor to remain in its functional mode and selects the Device Identification (ID) register to be connected between *TDI* and *TDO*. The Device ID register is a 32-bit shift register containing information regarding the IC manufacturer, device type, and version code. Accessing the Identification Register does not interfere with the operation of the processor. Also, access to the Identification Register is immediately available, via a TAP data scan operation, after power-up when the TAP has been reset with on-chip power-on or through the optional *TRST\_N* pin.

#### 8.4.3.3 IMPCODE Instruction

This instruction selects the Implementation register for output, which is always 32 bits.

#### 8.4.3.4 ADDRESS Instruction

This instruction is used to select the Address register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 32 bits through the *TDI* pin into the Address register and shifts out the captured address via the *TDO* pin.

#### 8.4.3.5 DATA Instruction

This instruction is used to select the Data register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 32 bits of *TDI* data into the Data register and shifts out the captured data via the *TDO* pin.

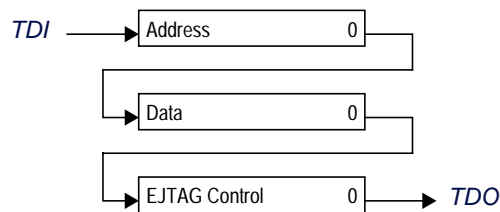
#### 8.4.3.6 CONTROL Instruction

This instruction is used to select the EJTAG Control register to be connected between *TDI* and *TDO*. The EJTAG Probe shifts 32 bits of *TDI* data into the EJTAG Control register and shifts out the EJTAG Control register bits via *TDO*.

#### 8.4.3.7 ALL Instruction

This instruction is used to select the concatenation of the Address and Data register, and the EJTAG Control register between *TDI* and *TDO*. It can be used in particular if switching instructions in the instruction register takes too many *TCK* cycles. The first bit shifted out is bit 0.

**Figure 8.23 Concatenation of the EJTAG Address, Data and Control Registers**



#### 8.4.3.8 EJTAGBOOT Instruction

When the EJTAGBOOT instruction is given and the Update-IR state is left, then the reset values of the ProbTrap, ProbEn and *EjtagBrk* bits in the *EJTAG Control* register are set to 1 after a hard or soft reset.

This EJTAGBOOT indication is effective until a NORMALBOOT instruction is given, *TRST\_N* is asserted or a rising edge of *TCK* occurs when the TAP controller is in Test-Logic-Reset state.

It is possible to make the CPU go into debug mode just after a hard or soft reset, without fetching or executing any instructions from the normal memory area. This can be used for download of code to a system which have no code in ROM.

The Bypass register is selected when the EJTAGBOOT instruction is given.

#### 8.4.3.9 NORMALBOOT Instruction

When the NORMALBOOT instruction is given and the Update-IR state is left, then the reset value of the ProbTrap, ProbEn and *EjtagBrk* bits in the *EJTAG Control* register are set to 0 after hard or soft reset.

The Bypass register is selected when the NORMALBOOT instruction is given.

#### 8.4.3.10 FASTDATA Instruction

This selects the Data and the Fastdata registers at once, as shown in [Figure 8.24](#).

**Figure 8.24 TDI to TDO Path When in Shift-DR State and FASTDATA Instruction is Selected**



#### 8.4.3.11 PCsample Register (PCSAMPLE Instruction)

This selects the PCsample Register. The use of the PCsample Register is described in more detail in [Section 8.9 “PC/Data Address Sampling”](#)

#### 8.4.3.12 FDC Instruction

This selects the Fast Debug Channel. The use of the FDC is described in more detail in [Section 8.10 “Fast Debug Channel”](#).

#### 8.4.3.13 TCBCONTROLA Instruction

This instruction is used to select the TCBCONTROLA register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

#### 8.4.3.14 TCBCONTROLB Instruction

This instruction is used to select the TCBCONTROLB register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register.

#### 8.4.3.15 TCBDATA Instruction

This instruction is used to select the TCBDATA register to be connected between *TDI* and *TDO*. This register is only implemented if the Trace Control Block is present. If no TCB is present, then this instruction will select the Bypass register. It should be noted that the TCBDATA register is only an access register to other TCB registers. The width of the TCBDATA register is dependent on the specific TCB register.

## 8.5 EJTAG TAP Registers

The EJTAG TAP Module has one Instruction register and a number of data registers, all accessible through the TAP:

### 8.5.1 Instruction Register

The Instruction register is accessed when the TAP receives an Instruction register scan protocol. During an Instruction register scan operation the TAP controller selects the output of the Instruction register to drive the *TDO* pin. The shift register consists of a series of bits arranged to form a single scan path between *TDI* and *TDO*. During an Instruction register scan operations, the TAP controls the register to capture status information and shift data from *TDI* to *TDO*. Both the capture and shift operations occur on the rising edge of *TCK*. However, the data shifted out from the

*TDO* occurs on the falling edge of *TCK*. In the Test-Logic-Reset and *Capture-IR* state, the instruction shift register is set to 00001<sub>2</sub>, as for the IDCODE instruction. This forces the device into the functional mode and selects the Device ID register. The Instruction register is 5 bits wide. The instruction shifted in takes effect for the following data register scan operation. A list of the implemented instructions are listed in [Table 8.28](#).

8.5.2 Data Registers Overview

The EJTAG uses several data registers, which are arranged in parallel from the primary *TDI* input to the primary *TDO* output. The Instruction register supplies the address that allows one of the data registers to be accessed during a data register scan operation. During a data register scan operation, the addressed scan register receives TAP control signals to capture the register and shift data from *TDI* to *TDO*. During a data register scan operation, the TAP selects the output of the data register to drive the *TDO* pin. The register is updated in the *Update-DR* state with respect to the write bits.

This description applies in general to the following data registers:

- Bypass Register
- Device Identification Register
- Implementation Register
- EJTAG Control Register (ECR)
- Processor Access Address Register
- Processor Access Data Register
- FastData Register

8.5.2.1 Bypass Register

The *Bypass* register consists of a single scan register bit. When selected, the Bypass register provides a single bit scan path between *TDI* and *TDO*. The Bypass register allows abbreviating the scan path through devices that are not involved in the test. The Bypass register is selected when the Instruction register is loaded with a pattern of all ones to satisfy the IEEE 1149.1 Bypass instruction requirement.

8.5.2.2 Device Identification (ID) Register

The *Device Identification* register is defined by IEEE 1149.1, to identify the device's manufacturer, part number, revision, and other device-specific information. [Table 8.29](#) shows the bit assignments defined for the read-only Device Identification Register, and inputs to the core determine the value of these bits. These bits can be scanned out of the *ID* register after being selected. The register is selected when the Instruction register is loaded with the IDCODE instruction.

Figure 8.25 Device Identification Register Format

31	28 27	12 11	1 0
Version	PartNumber	ManufID	R

**Table 8.29 Device Identification Register**

Fields		Description	Read/ Write	Reset State
Name	Bit(s)			
Version	31:28	<b>Version</b> (4 bits) This field identifies the version number of the processor derivative.	R	<i>EJ_Version[3:0]</i>
PartNumber	27:12	<b>Part Number</b> (16 bits) This field identifies the part number of the processor derivative.	R	<i>EJ_PartNumber[15:0]</i>
ManufID	11:1	<b>Manufacturer Identity</b> (11 bits) Accordingly to IEEE 1149.1-1990, the manufacturer identity code shall be a compressed form of the JEDEC Publications 106-A.	R	<i>EJ_ManufID[10:0]</i>
R	0	reserved	R	1

**8.5.2.3 Implementation Register**

This 32-bit read-only register is used to identify the features of the EJTAG implementation. Some of the reset values are set by inputs to the core. The register is selected when the Instruction register is loaded with the IMPCODE instruction.

**Figure 8.26 Implementation Register Format**

31	29	28	25	24	23	21	20	17	16	15	14	13	0
EJTAGver	reserved		DINT-sup	ASIDsize	reserved		MIPS16	0	NoDMA	reserved			

**Table 8.30 Implementation Register Descriptions**

Fields		Description	Read/Wr ite	Reset State						
Name	Bit(s)									
EJTAGver	31:29	EJTAG Version. 2: Version 2.6	R	5						
reserved	28:25	reserved	R	0						
DINTsup	24	DINT Signal Supported from Probe This bit indicates if the DINT signal from the probe is supported: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>DINT signal from the probe is not supported</td></tr><tr><td>1</td><td>Probe can use DINT signal to make debug interrupt.</td></tr></table>	Encoding	Meaning	0	DINT signal from the probe is not supported	1	Probe can use DINT signal to make debug interrupt.	R	<i>EJ_DINTsup</i>
Encoding	Meaning									
0	DINT signal from the probe is not supported									
1	Probe can use DINT signal to make debug interrupt.									

**Table 8.30 Implementation Register Descriptions**

Fields		Description	Read/Wr ite	Reset State										
Name	Bit(s)													
ASIDsize	23:21	Size of ASID field in implementation: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No ASID in implementation</td></tr><tr><td>1</td><td>6-bit ASID</td></tr><tr><td>2</td><td>8-bit ASID</td></tr><tr><td>3</td><td>Reserved</td></tr></table>	Encoding	Meaning	0	No ASID in implementation	1	6-bit ASID	2	8-bit ASID	3	Reserved	R	0
Encoding	Meaning													
0	No ASID in implementation													
1	6-bit ASID													
2	8-bit ASID													
3	Reserved													
reserved	20:17	reserved	R	0										
MIPS16	16	Indicates whether MIPS16 is implemented: <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No MIPS16 support</td></tr><tr><td>1</td><td>MIPS16 implemented</td></tr></table>	Encoding	Meaning	0	No MIPS16 support	1	MIPS16 implemented	R	0				
Encoding	Meaning													
0	No MIPS16 support													
1	MIPS16 implemented													
reserved	15	reserved	R	0										
NoDMA	14	No EJTAG DMA Support	R	1										
reserved	13:0	reserved	R	0										

**8.5.2.4 EJTAG Control Register**

This 32-bit register controls the various operations of the TAP modules. This register is selected by shifting in the *CONTROL* instruction. Bits in the *EJTAG Control* register can be set/cleared by shifting in data; status is read by shifting out the contents of this register. This *EJTAG Control* register can only be accessed by the TAP interface.

The *EJTAG Control* register is not updated in the *Update-DR* state unless the Reset occurred (Rocc) bit 31, is either 0 or written to 0. This is in order to ensure proper handling of processor accesses.

The value used for reset indicated in the table below takes effect on both hard and soft CPU resets, but not on TAP controller resets by e.g. *TRST\_N*. *TCK* clock is not required when the hard or soft CPU reset occurs, but the bits are still updated to the reset value when the *TCK* applies. The first 5 *TCK* clocks after hard or soft CPU resets may result in reset of the bits, due to synchronization between clock domains.

**Figure 8.27 EJTAG Control Register Format**

31	30	29	28	23	22	21	20	19	18	17	16	15	14	13	12	11	4	3	2	0
Rocc	Psz	Res	Doze	Halt	PerRst	PRnW	PrAcc	Res	PrRst	ProbEn	Prob-Trap	Res	Ejtag-Brk	Res	DM	Rs				



Table 8.31 EJTAG Control Register Descriptions

Fields		Description	Read/ Write	Reset State																																	
Name	Bit(s)																																				
Rocc	31	<p>Reset Occurred</p> <p>The bit indicates if a CPU reset has occurred:</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No reset occurred since bit last cleared.</td></tr><tr><td>1</td><td>Reset occurred since bit last cleared.</td></tr></table> <p>The <i>Rocc</i> bit will keep the 1 value as long as reset is applied.</p> <p>This bit must be cleared by the probe, to acknowledge that the incident was detected.</p> <p>The <i>EJTAG Control</i> register is not updated in the <i>Update-DR</i> state unless <i>Rocc</i> is 0, or written to 0. This is in order to ensure proper handling of processor access.</p>	Encoding	Meaning	0	No reset occurred since bit last cleared.	1	Reset occurred since bit last cleared.	R/W	1																											
Encoding	Meaning																																				
0	No reset occurred since bit last cleared.																																				
1	Reset occurred since bit last cleared.																																				
Psz[1:0]	30:29	<p>Processor Access Transfer Size</p> <p>These bits are used in combination with the lower two address bits of the Address register to determine the size of a processor access transaction. The bits are only valid when processor access is pending.</p> <table><tr><th>PAA[1:0]</th><th>Psz[1:0]</th><th>Transfer Size</th></tr><tr><td>00</td><td>00</td><td>Byte (LE, byte 0; BE, byte 3)</td></tr><tr><td>01</td><td>00</td><td>Byte (LE, byte 1; BE, byte 2)</td></tr><tr><td>10</td><td>00</td><td>Byte (LE, byte 2; BE, byte 1)</td></tr><tr><td>11</td><td>00</td><td>Byte (LE, byte 3; BE, byte 0)</td></tr><tr><td>00</td><td>01</td><td>Halfword (LE, bytes 1:0; BE, bytes 3:2)</td></tr><tr><td>10</td><td>01</td><td>Halfword (LE, bytes 3:2; BE, bytes 1:0)</td></tr><tr><td>00</td><td>10</td><td>Word (LE, BE; bytes 3, 2, 1, 0)</td></tr><tr><td>00</td><td>11</td><td>Triple (LE, bytes 2, 1, 0; BE, bytes 3, 2,1)</td></tr><tr><td>01</td><td>11</td><td>Triple (LE, bytes 3, 2, 1; BE, bytes 2, 1, 0)</td></tr><tr><td colspan="2">All others</td><td>Reserved</td></tr></table> <p>Note: LE=little endian, BE=big endian, the byte# refers to the byte number in a 32-bit register, where byte 3 = bits 31:24; byte 2 = bits 23:16; byte 1 = bits 15:8; byte 0=bits 7:0, independently of the endianness.</p>	PAA[1:0]	Psz[1:0]	Transfer Size	00	00	Byte (LE, byte 0; BE, byte 3)	01	00	Byte (LE, byte 1; BE, byte 2)	10	00	Byte (LE, byte 2; BE, byte 1)	11	00	Byte (LE, byte 3; BE, byte 0)	00	01	Halfword (LE, bytes 1:0; BE, bytes 3:2)	10	01	Halfword (LE, bytes 3:2; BE, bytes 1:0)	00	10	Word (LE, BE; bytes 3, 2, 1, 0)	00	11	Triple (LE, bytes 2, 1, 0; BE, bytes 3, 2,1)	01	11	Triple (LE, bytes 3, 2, 1; BE, bytes 2, 1, 0)	All others		Reserved	R	Undefined
PAA[1:0]	Psz[1:0]	Transfer Size																																			
00	00	Byte (LE, byte 0; BE, byte 3)																																			
01	00	Byte (LE, byte 1; BE, byte 2)																																			
10	00	Byte (LE, byte 2; BE, byte 1)																																			
11	00	Byte (LE, byte 3; BE, byte 0)																																			
00	01	Halfword (LE, bytes 1:0; BE, bytes 3:2)																																			
10	01	Halfword (LE, bytes 3:2; BE, bytes 1:0)																																			
00	10	Word (LE, BE; bytes 3, 2, 1, 0)																																			
00	11	Triple (LE, bytes 2, 1, 0; BE, bytes 3, 2,1)																																			
01	11	Triple (LE, bytes 3, 2, 1; BE, bytes 2, 1, 0)																																			
All others		Reserved																																			
Res	28:23	reserved	R	0																																	

Table 8.31 EJTAG Control Register Descriptions (Continued)

Fields		Description	Read/ Write	Reset State						
Name	Bit(s)									
Doze	22	<p>Doze state</p> <p>The Doze bit indicates any kind of low-power mode. The value is sampled in the Capture-DR state of the TAP controller:</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>CPU not in low-power mode.</td></tr><tr><td>1</td><td>CPU is in low-power mode.</td></tr></table> <p>Doze includes the Reduced Power (RP) and WAIT power-reduction modes.</p>	Encoding	Meaning	0	CPU not in low-power mode.	1	CPU is in low-power mode.	R	0
Encoding	Meaning									
0	CPU not in low-power mode.									
1	CPU is in low-power mode.									
Halt	21	<p>Halt state</p> <p>The Halt bit indicates if the internal system bus clock is running or stopped. The value is sampled in the Capture-DR state of the TAP controller:</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Internal system clock is running</td></tr><tr><td>1</td><td>Internal system clock is stopped</td></tr></table>	Encoding	Meaning	0	Internal system clock is running	1	Internal system clock is stopped	R	0
Encoding	Meaning									
0	Internal system clock is running									
1	Internal system clock is stopped									
PerRst	20	<p>Peripheral Reset</p> <p>When the bit is set to 1, it is only guaranteed that the peripheral reset has occurred in the system when the read value of this bit is also 1. This is to ensure that the setting from the <i>TCK</i> clock domain gets effect in the CPU clock domain, and in peripherals.</p> <p>When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is cleared in the CPU clock domain also.</p> <p>This bit controls the <i>EJ_PerRst</i> signal on the core.</p>	R/W	0						
PRnW	19	<p>Processor Access Read and Write</p> <p>This bit indicates if the pending processor access is for a read or write transaction, and the bit is only valid while PrAcc is set.</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Read transaction</td></tr><tr><td>1</td><td>Write transaction</td></tr></table>	Encoding	Meaning	0	Read transaction	1	Write transaction	R	Undefined
Encoding	Meaning									
0	Read transaction									
1	Write transaction									

Table 8.31 EJTAG Control Register Descriptions (Continued)

Fields		Description	Read/ Write	Reset State						
Name	Bit(s)									
PrAcc	18	<p>Processor Access (PA) Read value of this bit indicates if a Processor Access (PA) to the EJTAG memory is pending:</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>No pending processor access</td></tr><tr><td>1</td><td>Pending processor access</td></tr></table> <p>The probe's software must clear this bit to 0 to indicate the end of the PA. Write of 1 is ignored. A pending Processor Access is cleared when <i>ROCC</i> is set, but another PA may occur just after the reset if a debug exception occurs. Finishing a Processor Access is not accepted while the <i>ROCC</i> bit is set. This is to avoid that a Processor Access occurring after the reset is finished due to indication of a Processor Access that occurred before the reset. The FASTDATA access can clear this bit.</p>	Encoding	Meaning	0	No pending processor access	1	Pending processor access	R/W0	0
Encoding	Meaning									
0	No pending processor access									
1	Pending processor access									
Res	17	reserved	R	0						
PrRst	16	<p>Processor Reset (Implementation dependent behavior) When the bit is set to 1, then it is only guaranteed that this setting has taken effect in the system when the read value of this bit is also 1. This is to ensure that the setting from the <i>TCK</i> clock domain gets effect in the CPU clock domain, and in peripherals. When the bit is written to 0, then the bit must also be read as 0 before it is guaranteed that the indication is cleared in the CPU clock domain also. This bit controls the <i>EJ_PrRst</i> signal. If the signal is used in the system, then it must be ensured that both the processor and all devices required for a reset are properly reset. Otherwise the system may fail or hang. The bit resets itself, since the EJTAG Control register is reset by hard or soft reset.</p>	R/W	0						

Table 8.31 EJTAG Control Register Descriptions (Continued)

Fields		Description	Read/ Write	Reset State												
Name	Bit(s)															
ProbEn	15	<p>Probe Enable</p> <p>This bit indicates to the CPU if the EJTAG memory is handled by the probe so processor accesses are answered:</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>The probe does not handle EJTAG memory transactions</td></tr><tr><td>1</td><td>The probe does handle EJTAG memory transactions</td></tr></table> <p>It is an error by the software controlling the probe if it sets the ProbTrap bit to 1, but resets the <i>ProbEn</i> to 0. The operation of the processor is UNDEFINED in this case. The <i>ProbEn</i> bit is reflected as a read-only bit in the <i>ProbEn</i> bit, bit 0, in the <i>Debug Control Register (DCR)</i>.</p> <p>The read value indicates the effective value in the DCR, due to synchronization issues between <i>TCK</i> and CPU clock domains; however, it is ensured that change of the <i>ProbEn</i> prior to setting the <i>EjtagBrk</i> bit will have effect for the debug handler executed due to the debug exception.</p> <p>The reset value of the bit depends on whether the EJTAG-BOOT indication is given or not:</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Processor is in non-debug mode (No EJTAGBOOT indication given)</td></tr><tr><td>1</td><td>Processor is in debug mode (EJTAG-BOOT indication given)</td></tr></table>	Encoding	Meaning	0	The probe does not handle EJTAG memory transactions	1	The probe does handle EJTAG memory transactions	Encoding	Meaning	0	Processor is in non-debug mode (No EJTAGBOOT indication given)	1	Processor is in debug mode (EJTAG-BOOT indication given)	R/W	0 or 1 from EJTAGBOOT
Encoding	Meaning															
0	The probe does not handle EJTAG memory transactions															
1	The probe does handle EJTAG memory transactions															
Encoding	Meaning															
0	Processor is in non-debug mode (No EJTAGBOOT indication given)															
1	Processor is in debug mode (EJTAG-BOOT indication given)															

Table 8.31 EJTAG Control Register Descriptions (Continued)

Fields		Description	Read/Write	Reset State												
Name	Bit(s)															
ProbTrap	14	<p>Probe Trap</p> <p>This bit controls the location of the debug exception vector:</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>In normal memory 0xBFC0.0480</td></tr><tr><td>1</td><td>In EJTAG memory at 0xFF20.0200 in dmseg</td></tr></table> <p>Valid setting of the <i>ProbTrap</i> bit depends on the setting of the <i>ProbEn</i> bit, as described for the <i>ProbEn</i> bit.</p> <p>The <i>ProbTrap</i> should not be set to 1, for debug exception vector in EJTAG memory, unless the <i>ProbEn</i> bit is also set to 1 to indicate that the EJTAG memory may be accessed.</p> <p>The read value indicates the effective value to the CPU, due to synchronization issues between <i>TCK</i> and CPU clock domains; however, it is ensured that change of the <i>ProbTrap</i> bit prior to setting the <i>EjtagBrk</i> bit will have effect for the <i>EjtagBrk</i>.</p> <p>The reset value of the bit depends on whether the EJTAG-BOOT indication is given or not:</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Processor is in non-debug mode (No EJTAGBOOT indication given)</td></tr><tr><td>1</td><td>Processor is in debug mode (EJTAG-BOOT indication given)</td></tr></table>	Encoding	Meaning	0	In normal memory 0xBFC0.0480	1	In EJTAG memory at 0xFF20.0200 in dmseg	Encoding	Meaning	0	Processor is in non-debug mode (No EJTAGBOOT indication given)	1	Processor is in debug mode (EJTAG-BOOT indication given)	R/W	0 or 1 from EJTAGBOOT
Encoding	Meaning															
0	In normal memory 0xBFC0.0480															
1	In EJTAG memory at 0xFF20.0200 in dmseg															
Encoding	Meaning															
0	Processor is in non-debug mode (No EJTAGBOOT indication given)															
1	Processor is in debug mode (EJTAG-BOOT indication given)															
Res	13	reserved	R	0												
EjtagBrk	12	<p>EJTAG Break</p> <p>Setting this bit to 1 causes a debug exception to the processor, unless the CPU was in debug mode or another debug exception occurred.</p> <p>When the debug exception occurs, the processor <i>CoreType-1c</i> clock is restarted if the CPU was in low-power mode. This bit is cleared by hardware when the debug exception is taken.</p> <p>The reset value of the bit depends on whether the EJTAG-BOOT indication is given or not:</p> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Processor is in non-debug mode (No EJTAGBOOT indication given)</td></tr><tr><td>1</td><td>Processor is in debug mode (EJTAG-BOOT indication given)</td></tr></table>	Encoding	Meaning	0	Processor is in non-debug mode (No EJTAGBOOT indication given)	1	Processor is in debug mode (EJTAG-BOOT indication given)	R/W1	0 or 1 from EJTAGBOOT						
Encoding	Meaning															
0	Processor is in non-debug mode (No EJTAGBOOT indication given)															
1	Processor is in debug mode (EJTAG-BOOT indication given)															
Res	11:4	reserved	R	0												

Table 8.31 EJTAG Control Register Descriptions (Continued)

Fields		Description	Read/ Write	Reset State						
Name	Bit(s)									
DM	3	<div>Debug Mode</div> <div>This bit indicates the debug or non-debug mode:</div> <table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Processor is in non-debug mode</td></tr><tr><td>1</td><td>Processor is in debug mode</td></tr></table> <div>The bit is sampled in the <i>Capture-DR</i> state of the TAP controller.</div>	Encoding	Meaning	0	Processor is in non-debug mode	1	Processor is in debug mode	R	0
Encoding	Meaning									
0	Processor is in non-debug mode									
1	Processor is in debug mode									
Res	2:0	reserved	R	0						

### 8.5.3 Processor Access Address Register

The Processor Access Address (*PAA*) register is used to provide the address of the processor access in the dmseg, and the register is only valid when a processor access is pending. The length of the Address register is 32 bits, and this register is selected by shifting in the ADDRESS instruction.

#### 8.5.3.1 Processor Access Data Register

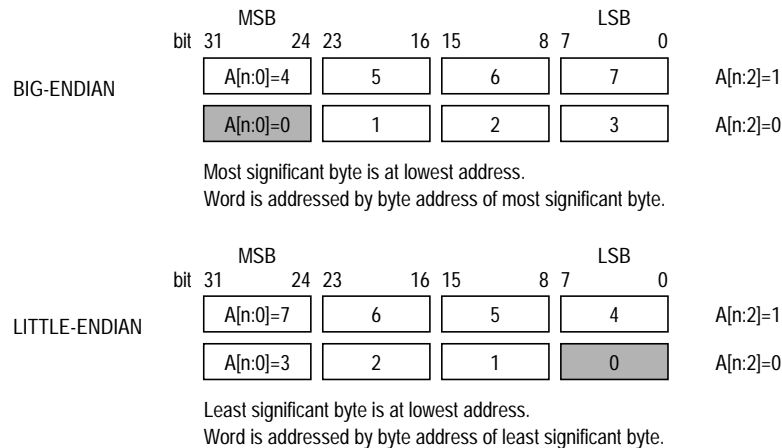
The *Processor Access Data (PAD)* register is used to provide data value to and from a processor access. The length of the Data register is 32 bits, and this register is selected by shifting in the DATA instruction.

The register has the written value for a processor access write due to a CPU store to the dmseg, and the output from this register is only valid when a processor access write is pending. The register is used to provide the data value for a processor access read due to a CPU load or fetch from the dmseg, and the register should only be updated with a new value when a processor access write is pending.

The *PAD* register is 32 bits wide. Data alignment is not used for this register, so the value in the *PAD* register matches data on the internal bus. The undefined bytes for a PA write are undefined, and for a *PAD* read then 0 (zero) must be shifted in for the unused bytes.

The organization of bytes in the *PAD* register depends on the endianness of the core, as shown in Figure 8.28. The endian mode for debug/kernel mode is determined by the state of the *SI\_Endian* input at power-up.

### Figure 8.28 Endian Formats for the PAD Register



The size of the transaction and thus the number of bytes available/required for the *PAD* register is determined by the *Psz* field in the *ECR*.

#### 8.5.4 Fastdata Register (TAP Instruction FASTDATA)

The width of the *Fastdata Register* is 1 bit. During a Fastdata access, the Fastdata register is written and read, i.e., a bit is shifted in and a bit is shifted out. During a Fastdata access, the Fastdata register value shifted in specifies whether the Fastdata access should be completed or not. The value shifted out is a flag that indicates whether the Fastdata access was successful or not (if completion was requested).

### Figure 8.29 Fastdata Register Format



### Table 8.32 Fastdata Register Field Description

Fields		Description	Read/ Write	Power-up State
Name	Bits			
SPrAcc	0	<p>Shifting in a zero value requests completion of the Fastdata access. The PrAcc bit in the EJTAG Control register is overwritten with zero when the access succeeds. (The access succeeds if PrAcc is one and the operation address is in the legal dmseg Fastdata area.) When successful, a one is shifted out. Shifting out a zero indicates a Fastdata access failure.</p> <p>Shifting in a one does not complete the Fastdata access and the PrAcc bit is unchanged. Shifting out a one indicates that the access would have been successful if allowed to complete and a zero indicates the access would not have successfully completed.</p>	R/W	Undefined

The FASTDATA access is used for efficient block transfers between dmseg (on the probe) and target memory (on the processor). An “upload” is defined as a sequence of processor loads from target memory and stores to dmseg. A “download” is a sequence of processor loads from dmseg and stores to target memory. The “Fastdata area” specifies the legal range of dmseg addresses (0xFF20.0000 - 0xFF20.000F) that can be used for uploads and downloads. The Data + Fastdata registers (selected with the FASTDATA instruction) allow efficient completion of pending Fastdata area accesses.

During Fastdata uploads and downloads, the processor will stall on accesses to the Fastdata area. The PrAcc (processor access pending bit) will be 1 indicating the probe is required to complete the access. Both upload and download accesses are attempted by shifting in a zero SPrAcc value (to request access completion) and shifting out SPrAcc to see if the attempt will be successful (i.e., there was an access pending and a legal Fastdata area address was used). Downloads will also shift in the data to be used to satisfy the load from dmseg’s Fastdata area, while uploads will shift out the data being stored to dmseg’s Fastdata area.

As noted above, two conditions must be true for the Fastdata access to succeed. These are:

- PrAcc must be 1, i.e., there must be a pending processor access.
- The Fastdata operation must use a valid Fastdata area address in dmseg (0xFF20.0000 to 0xFF20.000F).

Table 8.33 shows the values of the PrAcc and SPrAcc bits and the results of a Fastdata access.

**Table 8.33 Operation of the FASTDATA access**

Probe Operation	Address Match Check	PrAcc in the Control Register	LSB (SPrAcc) Shifted In	Action in the Data Register	PrAcc Changes To	LSB Shifted Out	Data Shifted Out
Download using FASTDATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	write data	0 (SPrAcc)	1	valid (previous) data
		0	x	none	unchanged	0	invalid
Upload using FASTDATA	Fails	x	x	none	unchanged	0	invalid
	Passes	1	1	none	unchanged	1	invalid
		1	0	read data	0 (SPrAcc)	1	valid data
		0	x	none	unchanged	0	invalid

There is no restriction on the contents of the Data register. It is expected that the transfer size is negotiated between the download/upload transfer code and the probe software. Note that the most efficient transfer size is a 32-bit word.

The Rocc bit of the Control register is not used for the FASTDATA operation.

## 8.6 TAP Processor Accesses

The TAP modules support handling of fetches, loads and stores from the CPU through the dmseg segment, whereby the TAP module can operate like a *slave unit* connected to the on-chip bus. The core can then execute code taken from the EJTAG Probe and it can access data (via a load or store) which is located on the EJTAG Probe. This occurs



in a serial way through the EJTAG interface: the core can thus execute instructions e.g. debug monitor code, without occupying the memory.

Accessing the dmseg segment (EJTAG memory) can only occur when the processor accesses an address in the range from 0xFF20.0000 to 0xFF2F.FFFF, the ProbEn bit is set, and the processor is in debug mode (DM=1). In addition the LSNM bit in the CP0 Debug register controls transactions to/from the dmseg.

When a debug exception is taken, while the ProbTrap bit is set, the processor will start fetching instructions from address 0xFF20.0200.

A pending processor access can only finish if the probe writes 0 to PrAcc or by a soft or hard reset.

### 8.6.1 Fetch/Load and Store from/to EJTAG Probe Through dmseg

1. The internal hardware latches the requested address into the PA Address register (in case of the Debug exception: 0xFF20.0200).
2. The internal hardware sets the following bits in the *EJTAG Control* register:  
*PrAcc* = 1 (selects Processor Access operation)  
*PRnW* = 0 (selects processor read operation)  
*Psz*[1:0] = value depending on the transfer size
3. The EJTAG Probe selects the *EJTAG Control* register, shifts out this control register's data and tests the *PrAcc* status bit (Processor Access): when the PrAcc bit is found 1, it means that the requested address is available and can be shifted out.
4. The EJTAG Probe checks the *PRnW* bit to determine the required access.
5. The EJTAG Probe selects the PA Address register and shifts out the requested address.
6. The EJTAG Probe selects the PA Data register and shifts in the instruction corresponding to this address.
7. The EJTAG Probe selects the *EJTAG Control* register and shifts a *PrAcc* = 0 bit into this register to indicate to the processor that the instruction is available.
8. The instruction becomes available in the instruction register and the processor starts executing.
9. The processor increments the program counter and outputs an instruction read request for the next instruction. This starts the whole sequence again.

Using the same protocol, the processor can also execute a load instruction to access the EJTAG Probe's memory. For this to happen, the processor must execute a load instruction (e.g. a LW, LH, LB) with the target address in the appropriate range.

Almost the same protocol is used to execute a store instruction to the EJTAG Probe's memory through dmseg. The store address must be in the range: 0xFF20.0000 to 0xFF2F.FFFF, the *ProbEn* bit must be set and the processor has to be in debug mode (DM=1). The sequence of actions is found below:

1. The internal hardware latches the requested address into the PA Address register
2. The internal hardware latches the data to be written into the PA Data register.

3. The internal hardware sets the following bits in the EJTAG Control register:  
*PrAcc* = 1 (selects Processor Access operation)  
*PRnW* = 1 (selects processor write operation)  
*PsZ*[1:0] = value depending on the transfer size
4. The EJTAG Probe selects the *EJTAG Control* register, shifts out this control register's data and tests the *PrAcc* status bit (Processor Access): when the *PrAcc* bit is found 1, it means that the requested address is available and can be shifted out.
5. The EJTAG Probe checks the *PRnW* bit to determine the required access.
6. The EJTAG Probe selects the PA Address register and shifts out the requested address.
7. The EJTAG Probe selects the PA Data register and shifts out the data to be written.
8. The EJTAG Probe selects the *EJTAG Control* register and shifts a *PrAcc* = 0 bit into this register to indicate to the processor that the write access is finished.
9. The EJTAG Probe writes the data to the requested address in its memory.
10. The processor detects that *PrAcc* bit = 0, which means that it is ready to handle a new access.

The above examples imply that no reset occurs during the operations, and that *Rocc* is cleared.

Note: Probe accesses and external bus accesses are serialized by the core. A probe access will not begin until all external bus requests have completed. Similarly, a new probe or external bus access will not begin until a pending probe access has completed.

## 8.7 SecureDebug

For security reasons, users can optionally disable certain EJTAG capabilities via the SecureDebug feature in order to prevent untrusted access to the core through debug mode.

### 8.7.1 Disabling EJTAG debugging

#### 8.7.1.1 EJ\_DisableProbeDebug Signal

An input signal to the core is defined, *EJ\_DisableProbeDebug*, which when asserted, forces *ProbEn*=0 and *ProbTrap*=0. *EJ\_DisableProbeDebug* overrides any other *ProbEn* or *ProbTrap* settings.

Suggested implementation of the EJ\_DisableProbeDebug signal is for a microcontroller to provide a bit within non-volatile memory (outside the core) that is pre-programmed to set or clear this control signal.

**Table 8.34 EJ\_DisableProbeDebug Signal Overview**

Signal	Description	Direction	Compliance
EJ_DisableProbeDebug	When asserted: <ul style="list-style-type: none"> <li>• ProbEn = 0</li> <li>• ProbTrap = 0</li> <li>• EjtagBrk is disabled<sup>1</sup>.</li> <li>• EJTAGBOOT is disabled.</li> <li>• PC Sampling is disabled.</li> <li>• DINT signal is ignored<sup>1</sup>.</li> </ul>	Input	Required for SecureDebug

1. An override is provided.

### 8.7.1.2 Override for EjtagBrk and DINT disable

An override for the EjtagBrk and DINT disable caused by the EJ\_DisableProbeDebug signal is provided by the Memory Protection Unit (MPU) *Config* register field *EjtagBrk\_Override*. This override feature is only available if the Memory Protection Unit is implemented.

Refer to the MPU chapter of “Security Features of the M14K™ Processor Family” (MD00896) for further information on the Memory Protection Unit.

The override can be asserted by the CPU during the trusted boot process. Its purpose is to allow a probe to assert EjtagBrk or the assertion of the DINT signal, which requests a Debug Interrupt exception, thus providing a means of recovering the CPU from a crash or hang. This feature allows a Debug Executive, if one is provided in target firmware, to communicate with the probe over the Fast Debug Channel (FDC) in order to get the attention of the target by causing a debug exception. It also allows a host-based debugger to query the target via Debug Executive commands to determine the cause of the hang.

## 8.7.2 EJTAG Features Unmodified by SecureDebug

SecureDebug will not modify the following EJTAG features:

- FDC (Fast Debug Channel) over EJTAG. This is required to provide a path for an EJTAG probe to send and receive messages via the Debug Executive when one is included in the target code. The physical EJTAG serial connection, pins, and protocol must function correctly as well as a cJTAG (2-wire) connection for FDC.
- RST\* signal. This is the hardware signal on the EJTAG connector that connects to the target system reset circuit. It can be asserted by an EJTAG probe.

## 8.8 iFlowtrace™ Mechanism

There is only one optional trace mechanism that is available to extract additional information about program execution. iFlowtrace is a light-weight instruction-only tracing scheme that is sufficient to reconstruct the execution flow in the core and it can only be controlled by debug software. This tracing scheme has been kept very simple to minimize the impact on die size.

The iFlowtrace tracing scheme is not a strict subset of the PDtrace tracing methodology, and its trace format outputs differ from those of PDtrace. Trace formats, using simplified instruction state descriptors, were designed for the iFlowtrace trace to simplify the trace mechanism and to obtain better compression.

Tracing is disabled if the processor enters Debug Mode (refer to the EJTAG specification for description of Debug Mode). This is true for both Normal Trace Mode as well as Special Trace Mode.

The presence of the iFlowtrace mechanism is indicated by the *CP0 Config3<sub>ITL</sub>* register bit.

### 8.8.1 A Simple Instruction-Only Tracing Scheme

A trace methodology can often be mostly defined by its inputs and outputs. Hence this basic scheme is described by the inputs to the core tracing logic and by the trace output format from the core. We assume here that the execution flow of the program is traced at the end of the execution path in the core similar to PDtrace.

#### 8.8.1.1 Trace Inputs

1. *In\_TraceOn*: when on, legal trace words are coming from the core and at the point when it is turned on, that is for the first traced instruction, a full PC value is output. When off, it cannot be assumed that legal trace words are available at the core interface.
2. *In\_Stall*: This says, stall the processor to avoid buffer overflow that can lose trace information. When off, a buffer overflow will simply throw away trace data and start over again. When on, the processor is signalled from the tracing logic to stall until the buffer is sufficiently drained and then the pipeline is restarted.

#### 8.8.1.2 Normal Trace Mode Outputs

1. Stall cycles in the pipe are ignored by the tracing logic and are not traced. This is indicated by the signal *Out\_Valid* that is turned off when no valid instruction is being traced. When *Out\_Valid* is asserted, instructions are traced out as described in the rest of this section. The traced instruction PC is a virtual address.
2. In the output format, every sequentially executed instruction is traced as 1'b0.
3. Every instruction that is not sequential to the previous one is traced as either a 10 or an 11 (read this as a serial bitstream from left to right). This implies that the target instruction of a branch or jump is traced this way, not the actual branch or jump instruction (this is similar to PDtrace):
4. A 10 instruction implies a taken branch for a conditional branch instruction whose condition is unpredictable statically, but whose branch target can be computed statically and hence the new PC does not need to be traced out. Note that if this branch was not taken, it would have been indicated by a 0 bit, that is sequential flow.
5. A 11 instruction implies a taken branch for an indirect jump-like instruction whose branch target could not be computed statically and hence the taken branch address is now given in the trace. This includes, for example, instructions like jr, jalr, and interrupts:
  - 11 00 - followed by 8 bits of 1-bit shifted offset from the last PC. The bit assignments of this format on the bus between the core tracing logic and the ITCB is:

[3:0] = 4'b0011

[11:4] = PCdelta[8:1]

- 11 01 - followed by 16 bits of 1-bit shifted offset from the last PC. The bit assignments of this format on the bus between the core tracing logic and the ITCB is:

[3:0] = 4'b1011  
[19:4] = PCdelta[16:1]

- 11 10 - followed by 31 of the most significant bits of the PC value, followed by a bit (NCC) that indicates no code compression. Note that for a MIPS32 or MIPS64 instruction, NCC=1, and for microMIPS instruction NCC=0. This trace record will appear at all transition points between MIPS32/MIPS64 and microMIPS instruction execution.

This form is also a special case of the 11 format and it is used when the instruction is not a branch or jump, but nevertheless the full PC value needs to be reconstructed. This is used for synchronization purposes, similar to the Sync in PDtrace. In iFlowtrace rev 2.0 onwards, the sync period is user-defined, and is counted down and when an internal counter runs through all the values, this format is used. The bit assignments of this format on the bus between the core tracing logic and the ITCB is:

[3:0] = 4'b0111  
[34:4] = PC[31:1]  
[35] = NCC

- 11 11 - Used to indicate trace resumption after a discontinuity occurred. The next format is a 1110 that sends a full PC value. A discontinuity might happen due to various reasons, for example, an internal buffer overflow, and at trace-on/trace-off trigger action.

## 8.8.2 Special Trace Modes

iFlowtrace 2.0 adds special trace modes which can only be active when the normal tracing mode is disabled. Software can determine which modes are supported by attempting to write the enable bits in the *IFCTL* register. Software can check the Illegal bit in the *IFCTL* register—if an unsupported combination of modes is requested, the bit will be set and the trace contents will be unpredictable. The special trace modes are described below.

### 8.8.2.1 Mode Descriptions

#### ***Delta Cycle Mode***

This mode is specified in combination with the other special trace modes. It is enabled via the CYC bit in the Control/Status Register. When delta cycle reporting is enabled, each trace message will include a 10b delta cycle value which reports the number of cycles that have elapsed since the last message was generated. A value of 0 indicates that the two messages were generated in the same cycle. A value of 1 indicates that they were generated in consecutive cycles. If 1023 cycles elapse without an event being traced, a counter rollover message is generated.

Note: If the processor clocks stop due to execution of the WAIT instruction, the delta cycle counter will also stop and will report 'active' cycles between events rather than 'total' cycles.

#### ***Breakpoint Match Mode***

This mode uses EJTAG data and instruction breakpoint hardware to enable a trace of PC values. Instead of starting or stopping trace, a triggerpoint will cause a single breakpoint match trace record. This record indicates that there was a triggerpoint match, the breakpoint ID of the matching breakpoint, and the PC value of an instruction that matched the instruction of data breakpoint. This mode can only be used when normal tracing mode is turned off. This mode can

not be used in conjunction with other special trace modes. This mode is enabled or disabled via the BM field in the Control/Status register (see [Section 8.8.6 “ITCB Register Interface for Software Configurability”](#)).

The breakpoints used in this mode must have the TE bit set to enable the match condition.

Software should avoid setting up overlapping breakpoints. The behavior when multiple matches occur on the same instruction is to report a BreakpointID of 7.

### ***Filtered Data Tracing Mode***

This mode uses EJTAG data breakpoint hardware to enable a trace of data values. Rather than starting or stopping trace as in normal trace mode, a data triggerpoint will cause a filtered data trace record. This record indicates that there was a data triggerpoint match, the breakpoint ID of the matching breakpoint, whether it was a load or store, the size of the request, low order address bits, and the data value. This mode can only be used when normal tracing mode is turned off. This mode can not be used in conjunction with other special trace modes. This mode can be enabled or disabled via the FDT bit in the Control/Status register (see [Section 8.8.6 “ITCB Register Interface for Software Configurability”](#)).

The corresponding data breakpoint must have the TE bit set to enable the match condition.

Software should avoid setting up overlapping data breakpoints. The behavior when multiple matches on one load or store are detected is to report a BreakpointID of 7.

### ***Extended Filtered Data Tracing Mode***

Extends Filtered Data Tracing Mode by adding the virtual address of the load/store instruction to the generated trace information. (see [Section “Filtered Data Tracing Mode”](#) above).

This behavior is enabled/disabled by the FDT\_CAUSE field in the *IFCTL* Control/Status register (see [Section 8.8.6 “ITCB Register Interface for Software Configurability”](#)). FDT\_CAUSE only has effect if the FDT field is also set.

The extended trace sequence is a FDT trace message followed by the Breakpoint Match (BM) trace message. If the *IFCTL*<sub>CYC</sub> field is set, the FDTtrace message will have a DeltaCycle Message value of ‘0’ directly followed by the Breakpoint Match message. This message sequence (FDT, delta cycle of 0, and BM) indicates to the trace disassembler that Extended Filtered Data Tracing mode is enabled (*IFCTL*<sub>FDT\_CAUSE</sub>=1).

### ***Function Call/Return and Exception Tracing Mode***

In this mode, the PC value of function calls and returns and/or exceptions and returns are traced out. This mode can only be used when normal tracing mode is turned off. This mode cannot be used in conjunction with other special trace modes. The function call/return and exception/return are independently enabled or disabled via the FCR and ER bits in the Control/Status register (see [Section 8.8.6 “ITCB Register Interface for Software Configurability”](#)).

These events are reported for the following instructions:

- MIPS32 function calls: JAL, JALR, JALR.HB, JALX
- microMIPS function calls: JAL, JALR, JALR.HB, JALX, JALR16, JALRS16, JALRS, JALRS.HB, JALS
- MIPS32 function returns: JR, JR.HB
- microMIPS function returns: JR, JR.HB, JRC, JRADDIUSP, JR16

- Exceptions: Reported on the first instruction of the exception handler
- Exception returns: ERET
- MCU ASE Interrupt returns: IRET

### Other Trace Messages

In any of the special trace modes, it is possible to embed messages into the trace stream directly from a program. This is done by writing to the *UserTraceData1* or *UserTraceData2* Cop0 registers. When *UserTraceData1* register is written, a trace message of type “User Triggered Message 1” (UTM1) is generated. When *UserTraceData2* register is written, a trace message of type “User Triggered Message 2” (UTM2) is generated. Please refer to 5.2.30 “User Trace Data1 Register (CP0 Register 23, Select 3)/User Trace Data2 Register (CP0 Register 24, Select 3)” on page 141.

Overflow messages can also be generated when tracing off-chip if the IO control bit is 0 and trace data is generated faster than it is consumed. No overflow will be generated when using on-chip trace.

#### 8.8.2.2 Special Trace Mode Outputs

The normal and special trace modes cannot be enabled at the same time because the trace message encoding is not unique between the two modes. The software reading the trace stream must be aware of which mode is selected to know how to interpret the bits in the trace stream. The message types for each type of special trace message are unique.

- 00 (as above, read a bitstream from left to right) - Delta Cycle Rollover message. The output format is:  
[1:0] = 2'b00
- 010 - User Trace Message. The format of this type of message is:  
[2:0] = 3'b010  
[34:3] = Data[31:0]  
[35] = UTM2/UTM1 (1=UTM2, 0=UTM1)  
[44:36] = DeltaCycle (if enabled)
- 011 - Reserved
- 10 - Breakpoint Match Message. The output format during this trace mode is:  
[1:0] = 2'b01  
[5:2] = BreakpointID  
[6] = Instruction Breakpoint  
[37:7] = MatchingPC[31:1]  
[38] = NCC  
[48:39] = DeltaCycle (if enabled)  
Note that for a MIPS32 or MIPS64 instruction, NCC=1, and for microMIPS instruction NCC=0.
- 110 - Filtered Data Message. The output format during this trace mode is:  
[2:0] = 3'b011  
[6:3] = BreakpointID  
[7] = Load/Store (1=Load, 0=Store)  
[8] = FullWord (1=32b data, 0=<32b)  
[14:5] = Addr[7:2]  
[46:15] = { 32b data value } OR  
[46:15] = { BE[3:0], 4'b0, 24b data value } OR  
[46:15] = { BE[3:0], 12'b0, 16b data value } OR

[46:15] = {BE[3:0], 20'b0, 8b data value}

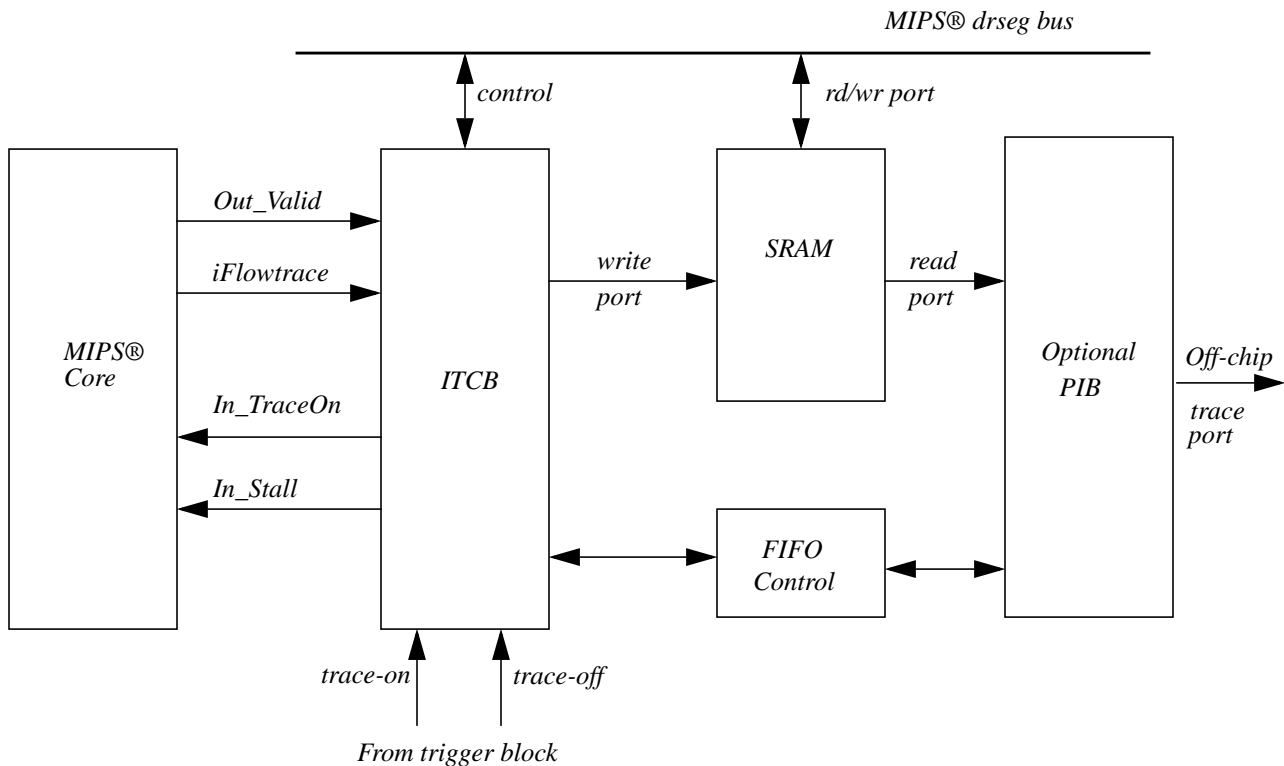
[56:47] = DeltaCycle (if enabled)

- 1110 - Function Call/Return/Exception Tracing. The output format during this trace mode is:  
 [3:0] = 4'b0111  
 [4] = FC  
 [5] = Ex  
 [6] = R  
 [37:8] = PC[31:1]  
 [38] = NCC  
 [48:39] = Delta Cycle (if enabled)  
 Note that for a MIPS32 or MIPS64 instruction, NCC=1, and for microMIPS instruction NCC=0. FC=1 implies a function call, Ex=1 implies the start of an exception handler, and R=1 implies a function or exception return.
- 1111- Overflow message. The format of this type of message is:  
 [3:0] = 4'b1111

## 8.8.3 ITCB Overview

The IFlowTrace Control Block (ITCB) is responsible for accepting trace signals from the CPU core, formatting them, and storing them into an on-chip FIFO. The figure also shows the Probe Interface Block (PIB) which reads the FIFO and outputs the memory contents through a narrow off-chip trace port.

**Figure 8.30 Trace Logic Overview**





### 8.8.4 ITCB IFlowTrace Interface

The iFlowtrace interface consists of 57 data signals plus a valid signal. The 57 data signals encode information about what the CPU is doing in each clock cycle. Valid indicates that the CPU is executing an instruction in this cycle and therefore the 57 data signals carry valid execution information. The iFlowtrace data bus is encoded as shown in [Table 8.35](#). Note that all the non-defined upper bits of the bus are zeroes.

**Table 8.35 Data Bus Encoding**

Valid	Data (LSBs)	Description
0	X	No instructions executed in this cycle
1	0	Normal Mode: Sequential instruction executed
1	01	Normal Mode: Branch executed, destination predictable from code
1	<8>0011	Normal Mode: Discontinuous instruction executed, PC offset is 8 bit signed offset
1	<16>1011	Normal Mode: Discontinuous instruction executed, PC offset is 16 bit signed offset
1	<NCC><31>0111	Normal Mode: Discontinuous instruction or synchronization record, No Code Compression (NCC) bit included as well as 31 MSBs of the PC value
1	00	Special Mode: Delta Cycle Rollover message
1	<10><32>010	Special Mode: User add-in Trace Message. 32 bit user data as well as 10 bit delta cycle if enabled.
1	<10><NCC><31><1><4>01	Special Mode: Breakpoint Match Message. 4 bit breakpoint ID, 1 bit indicate breakpoint type, 31 MSBs of the PC value, NCC bit included as well as 10 bit delta cycle if enable.
1	<10><32><6><1><1><4>011	Special Mode: Filtered Data Message. 4 bit breakpoint ID, 1 bit load or store indication, 1 bit full word indication, 6 bit of addr[7:2], 32 bit of the data information included as well as 10 bit delta cycle if enabled.
1	<10><NCC><31><R><Ex><FC>011	Special Mode: Function Call/Return/Exception Tracing. 1 bit function call indication, 1 bit exception indication, 1 bit function or exception return indication, 31 MSBs of the PC value, NCC bit included as well as 10 bit delta cycle if enabled.
1	1111	Internal overflow

### 8.8.5 TCB Storage Representation

Records from iFlowtrace are inserted into a memory stream exactly as they appear in the iFlowtrace data output. Records are concatenated into a continuous stream starting at the LSB. When a trace word is filled, it is written to memory along with some tag bits. Each record consists of a 64-bit word, which comprises 58 message bits and 6 tag bits or header bits that clarify information about the message in that word.

The ITCB includes a 58-bit shift register to accumulate trace messages. Once 58 or more bits are accumulated, the 58 bits and 6 tag bits are sent to the memory write interface. Messages may span a trace word boundary. In this case, the 6 tag bits indicate the bit number of the first full trace message in the 58-bit data field.

The tag bits are slightly encoded so they can serve a secondary purpose of indicating to off-chip trace hardware when a valid trace word transmission begins. The encoding ensures that at least one of the 4 LSBs of the tag is always a 1 for a valid trace message. The tag values are shown in [Table 8.36](#). The longest trace message is 57 bits (filtered data

trace in special trace mode with delta cycle), so the starting position indicated by the tag bits is always between 0 and 56.

**Table 8.36 Tag Bit Encoding**

Starting Bit of First Full Trace Message	Encoding (decimal)
0	58
16	59
32	60
48	61
Unused	0,16,32,48
Reserved	62,63
Others	StartingBit

When trace stops (ON set to zero), any partially filled trace words are written to memory. Any unused space above the final message is filled with 1's. The decoder distinguishes 1111 patterns used for fill in this position from an 1111 overflow message by recognizing that it is the last trace word.

These trace formats are written to a trace memory that is either on-chip or off-chip. No particular size of SRAM is specified; the size is user selectable based on the application needs and area trade-offs. Each trace word can typically store about 20 to 30 instructions in normal trace mode, so a 1 KWord trace memory could store the history of 20K to 30K executed instructions.

The on-chip SRAM or trace memory is written continuously as a circular buffer. It is accessible via drseg address mapped registers. There are registers for the read pointer, write pointer, and trace word. The write pointer register includes a wrap bit that indicates that the pointer has wrapped since the last time the register was written. Before starting trace, the write pointer would typically be set to 0. To read the trace memory, the read pointer should be set to 0 if there has not been a wrap, or to the value of the write pointer if there has been. Reading the trace word register will read the entry pointed to by the read pointer and will automatically increment the read pointer. Software can continue reading until all valid entries have been read out.

## 8.8.6 ITCB Register Interface for Software Configurability

The ITCB includes a drseg memory interface to allow software to set up tracing and read the current status. If an on-chip trace buffer is also implemented, there are additional registers included for accessing it.

### 8.8.6.1 IFlowTrace Control/Status (IFCTL) Register (offset 0x3fc0)

The Control/Status register provides the mechanism for turning on the different trace modes. [Figure 8.31](#) has the format of the register and [Table 8.37](#) describes the register fields.

**Figure 8.31 Control/Status Register**

31	30	16	15	14	13	12	11	10	9	8	5	4	3	2	1	0
Illegal	0	FDT_CAUSE	CYC	FDT	BM	ER	FCR	EST	SyP	OfClk	OfC	IO	En	On		

Table 8.37 Control/Status Register Field Descriptions

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
0	30:16	Reserved for future use. Read as zeros, must be written as zeros	R	0	Required
Illegal	31	This bit is set by hardware and indicates if the currently enabled trace output modes are an illegal combination. A value of 1 indicates an unsupported setting. A value of 0 indicates that the currently selected settings are legal.	R	0	Required
FDT_CAUSE	15	Extended Filtered Data Trace mode (FDT). Adds causing load/store virtual address to Filtered Data Trace. FDT_CAUSE only has effect if FDT is set. The extended trace sequence is a FDT trace message followed by the Breakpoint Match (BM) trace message. If CYC is set, the FDT trace message will have a DeltaCycle Message value of '0' directly followed by the Breakpoint match (BM) message. This message sequence (FDT, delta cycle of 0, and BM) indicates to the trace disassembler that Extended Filtered Data Tracing mode is enabled.	R/W	0	Optional for iFlowTrace rev 2.0+
CYC	14	Delta Cycle Mode: This mode can be set in combination with the EST special trace modes. When set, a delta cycle value is included in each of the trace messages and indicates the number of cycles since the last message was generated. If this tracing mode is not implemented, the field is read-only and read as zero.	R/W	0	Optional for iFlowTrace rev 2.0+
FDT	13	Filtered Data Trace mode. If set, on a data breakpoint match, the data value of the matching breakpoint is traced. Normal tracing is inhibited when this mode is active. If this tracing mode is not implemented, the field is read-only and read as zero.	R/W	0	Optional for iFlowTrace rev 2.0+
BM	12	Breakpoint Match. If set, only instructions that match instruction or data breakpoints are traced. Normal tracing is inhibited when this mode is active. If this tracing mode is not implemented, the field is read-only and read as zero.	R/W	0	Optional for iFlowTrace rev 2.0+
ER	11	Trace exceptions and exception returns. If set, trace includes markers for exceptions and exception returns. Can be used in conjunction with the FCR bit. Inhibits normal tracing. If this tracing mode is not implemented, the field is read-only and read as zero.	R/W	0	Optional for iFlowTrace rev 2.0+
FCR	10	Trace Function Calls and Returns. If set, trace includes markers for function calls and returns. Can be used in conjunction with the ER bit. If this tracing mode is not implemented, the field is read-only and read as zero.	R/W	0	Optional for iFlowTrace rev 2.0+
EST	9	Enable Special Tracing Modes. If set, normal tracing is inhibited, allowing the user to choose one of several special tracing modes. Setting this bit inhibits normal trace mode. If no special tracing modes are implemented, this field is read-only, and read as zero.	R/W	0	Optional for iFlowTrace rev 2.0+
SyP	8:5	Synchronization Period. The synchronization period is set to $2^{(SyP+8)}$ instructions. Thus a value of 0x0 implies 256 instructions, and a value of 0xF implies 8M instructions.	R/W	0	Required for iFlowTrace rev 2.0+

Table 8.37 Control/Status Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State	Compliance
Name	Bits				
OfClk	4	Controls the Off-chip clock ratio. When the bit is set, this implies 1:2, that is, the trace clock is running at 1/2 the core clock, and when the bit is clear, implies 1:4 ratio, that is, the trace clock is at 1/4 the core clock. Ignored unless OfC is also set.	R/W	0	Required
OfC	3	Off-chip. 1 enables the PIB (if present) to unload the trace memory. 0 disables the PIB and would be used when on-chip storage is desired or if a PIB is not present. This bit is settable only if the design supports both on-chip and off-chip modes. Otherwise is a read-only bit indicating which mode is supported.	R/W or R	Preset	Required
IO	2	Inhibit overflow. If set, the CPU is stalled whenever the trace memory is full. Ignored unless OfC is also set.	R/W	0	Required
En	1	Trace enable. This bit may be set by software or by Trace-on/Trace-off action bits from the Complex Trigger block. Software writes EN with the desired initial state of tracing when the ITCB is first turned on and EN is controlled by hardware thereafter. EN turning on and off does not flush partly filled trace words.	R/W	0	Required
On	0	Software control of trace collection. 0 disables all collection and flushes out any partially filled trace words.	R/W	0	Required

#### 8.8.6.2 ITCBTW Register (offset 0x3F80)

The *ITCBTW* register is used to read Trace Words from the on-chip trace memory. The TW read is the TW pointed to by the *ITCBRDP* register. A side effect of reading the *ITCBTW* register is that the *ITCBRDP* register increments to the next TW in the on-chip trace memory. If *ITCBRDP* is at the max size of the on-chip trace memory, the increment wraps back to address zero.

Note that this is a 64b register. On a 32b processor, software must read the upper word (offset 0x3F84) first as the address increment takes place on a read of the lower word (0x3F80).

The format of the *ITCBTW* register is shown below, and the field is described in [Table 8.38](#).

Figure 8.32 ITCBTW Register Format

63	0
Data	

Table 8.38 ITCBTW Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Names	Bits				
Data	63:0	Trace Word	R	Undefined	Required

### 8.8.6.3 ITCBRDP Register (Offset 0x3f88)

The *ITCBRD*P register is the address pointer to on-chip trace memory. It points to the TW read when reading the *ITCBTW* register. This value will be automatically incremented after a read of the *ITCBTW* register.

The format of the *ITCBRD*P register is shown below, and the field is described in Table 8.39. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

**Figure 8.33 ITCBRDP Register Format**



**Table 8.39 ITCBRDP Register Field Descriptions**

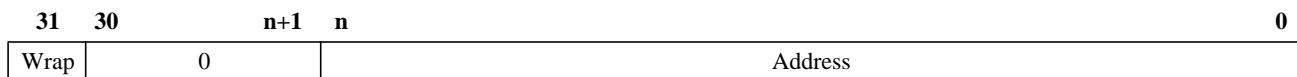
Fields		Description	Read/ Write	Reset State	Compliance
Names	Bits				
Data	31:(n+1)	Reserved. Must be written zero, reads back zero.	0	0	Required
Address	n:0	Byte address of on-chip trace memory word.	R/W	Undefined	Required

### 8.8.6.4 ITCBWRP Register (Offset 0x3f90)

The *ITCBWR*P register is the address pointer to on-chip trace memory. It points to the location where the next new TW for on-chip trace will be written. The top bit in the register indicates whether the pointer has wrapped. If it has, then the write pointer will also point to the oldest trace word, and the read pointer can be set to that to read the entire array in order. If it is cleared, then the read pointer can be set to 0 to read up to the write pointer position.

The format of the *ITCBWR*P register is shown below, and the field is described in Table 8.40. The value of n depends on the size of the on-chip trace memory. As the address points to a 64-bit TW, lower three bits are always zero.

**Figure 8.34 ITCBWRP Register Format**



**Table 8.40 ITCBWRP Register Field Descriptions**

Fields		Description	Read/ Write	Reset State	Compliance
Names	Bits				
Wrap	31	Indicates that the entire array has been written at least once	R/W	Undefined	Required
0	30:(n+1)	Reserved. Must be written zero, reads back zero.	0	0	Required
Address	n:0	Byte address of the next on-chip trace memory word to be written	R/W	Undefined	Required

## 8.8.7 ITCB iFlowtrace Off-Chip Interface

The off-chip interface consists of a 4-bit data port (*TR\_DATA*) and a trace clock (*TR\_CLK*). *TR\_CLK* can be a DDR clock; that is, both edges are significant. *TR\_DATA* and *TR\_CLK* follow the same timing and have the same output structure as the PDtrace TCB described in MIPS specifications. The trace clock is synchronous to the system clock but running at a divided frequency. The *OfClk* bit in the *Control/Status* register indicates the ratio between the trace clock and the core clock. The Trace clock is always 1/2 of the trace port data rate, hence the “full speed” ITCB outputs data at the CPU core clock rate but the trace clock is half that, hence the 1:2 *OfClk* value is the full speed, and the 1:4 *OfClk* ratio is half-speed.

When a 64-bit trace word is ready to transmit, the PIB reads it from the FIFO and begins sending it out on *TR\_DATA*. It is sent in 4-bit increments starting at the LSBs. In a valid trace word, the 4 LSBs are never all zero, so a probe listening on the *TR\_DATA* port can easily determine when the transmission begins and then count 15 additional cycles to collect the whole 64-bit word. Between valid transmissions, *TR\_DATA* is held at zero and *TR\_CLK* continues to run.

*TR\_CLK* runs continuously whenever a probe is connected. An optional signal *TR\_PROBE\_N* may be pulled high when a probe is not connected and could be used to disable the off-chip trace port. If not present, this signal must be tied low at the Probe Interface Block (PIB) input.

The following encoding is used for the 6 tag bits to tell the PIB receiver that a valid transmission is starting:

```
// if (srcount == 0), EncodedSrCount = 111010 = 58
// else if (srcount == 16) EncodedSrCount = 111011 = 59
// else if (srcount == 32) EncodedSrCount = 111100 = 60
// else if (srcount == 48) EncodedSrCount = 111101 = 61
// else EncodedSrCount = srcount
```

## 8.8.8 Breakpoint-Based Enabling of Tracing

Each hardware breakpoint in the EJTAG block (see the MIPS EJTAG Specification, MD00047, revision 4.14) has a control bit associated with it that enables a trigger signal to be generated on a break match condition. In special trace mode, this trigger can be used to insert an event record into the trace stream. In normal trace mode, this trigger signal can be used to turn trace on or off, thus allowing a user to control the trace on/off functionality using breakpoints. Similar to the TraceIBPC and TraceDBPC registers in PDtrace, registers are defined to control the start and stop of iFlowtrace. The details on the actual register names and drseg addresses are shown in [Table 8.41](#).

**Table 8.41 drseg Registers that Enable/Disable Trace from Breakpoint-Based Triggers**

Register Name	drseg Address	Reset Value	Description
ITrigiFlowTrcEn	0x3FD0	0	Register that controls whether or not hardware instruction breakpoints can trigger iFlowtrace tracing functionality
DTrigiFlowTrcEn	0x3FD8	0	Register that controls whether or not hardware data and tuple breakpoints can trigger iFlowtrace tracing functionality

The bits in each register are defined as follows:

- Bit 28 (IE/DE): Used to specify whether the trigger signal from EJTAG simple or complex instruction (data or tuple) break should trigger iFlowTrace tracing functions or not. A value of 0 disables trigger signals from EJTAG instruction breaks, and 1 enables triggers for the same.

- Bits 14:0 (IBrk/DBrk): Used to explicitly specify which instruction (data or tuple) breaks enable or disable iFlowTrace. A value of 0 implies that trace is turned off (unconditional trace stop) and a value of 1 specifies that the trigger enables trace (unconditional trace start).

## 8.9 PC/Data Address Sampling

It is often useful for program profiling and analysis to periodically sample the value of the PC. This information can be used for statistical profiling akin to gprof, and is also very useful for detecting hot-spots in the code. In a multi-threaded environment, this information can be used to understand thread behavior, and to verify thread scheduling mechanisms in the absence of a full-fledged tracing facility like PDtrace.

The PC sampling feature is optional within EJTAG, but EJTAG and the TAP controller must be implemented if PC Sampling is required. When implemented, PC sampling can be turned on or off using an enable bit; when the feature is enabled, the PC value is continually sampled.

The presence or absence of the PC Sampling feature is indicated by the *PCS* (PC Sample) bit in the Debug Control Register. If PC sampling is implemented, and the *PCSe* (PC Sample Enable) bit in the Debug Control Register is also set to one, then the PC values are constantly sampled at the defined rate ( $DCR_{PCR}$ ) and written to a TAP register. The old value in the TAP register is overwritten by the new value, even if this register has not been read out by the debug probe.

The presence or absence of Data Address Sampling is indicated by the *DAS* (Data Address Sample) bit in the Debug Control Register and enabled by the *DASe* (Data Address Sampling Enable) bit in the Debug Control Register.

The sample rate is specified by the 3-bit *PCR* (PC Sample Rate) field (bits 8:6) in the Debug Control Register (*DCR*). These three bits encode a value  $2^5$  to  $2^{12}$  in a manner similar to the specification of SyncPeriod. When the implementation allows these bits to be written, the internal PC sample counter will be reset by each write, so that counting for the requested sample rate is immediately restarted.

The sample format includes a New data bit, the sampled value, the ASID of the sampled value (if not disabled by PCnoASID, bit 25 in *DCR*). Figure 8.35 shows the format of the sampled values in the PCSAMPLE TAP register for MIPS32. The New data bit is used by the probe to determine if the sampled data just read out is new or has already been read and must be discarded.

**Figure 8.35 PCSAMPLE TAP Register Format (MIPS32)**

40	33	32	1	0
ASID (if enabled)	PC or Data Address			New

The sampled PC value is the PC of the graduating instruction in the current cycle. If the processor is stalled when the PC sample counter overflows, then the sampled PC is the PC of the next graduating instruction. The processor continues to sample the PC value even when it is in Debug mode.

Note that some of the smaller sample periods can be shorter than the time needed to read out the sampled value. That is, it might take 41 (TCK) clock ticks to read a MIPS32 sample, while the smallest sample period is 32 (processor) clocks. While the sample is being read out, multiple samples may be taken and discarded, needlessly wasting power. To reduce unnecessary overhead, the TAP register includes only those fields that are enabled. If both PC Sampling and Data Sampling are enabled, then both samples are included in the PCSample scan register. PC Sample is in the least significant bits followed by a Data Address Sample. If either PC Sampling or Data Address Sampling is disabled, then the TAP register does not include that sample. The total scan length is  $49 * 2 = 82$  bits if all fields are present and enabled.

### 8.9.1 PC Sampling in Wait State

Note that the processor samples PC even when it is asleep, that is, in a WAIT state. This permits an analysis of the amount of time spent by a processor in WAIT state which may be used for example to revert to a low power mode during the non-execution phase of a real-time application. But counting cycles to update the PC sample value is a waste of power. Hence, when in a WAIT state, the processor must simply switch the New bit to 1 each time it is set to 0 by the probe hardware. Hence, the external agent or probe reading the PC value will detect a WAIT instruction for as long as the processor remains in the WAIT state. When the processor leaves the WAIT state, then counting is resumed as before.

### 8.9.2 Data Address Sampling

EJTAG revision 5.0 extends the PC sampling mechanism to allow sampling of data (load and store) addresses. This feature is enabled with DASE, bit 23 in the Debug Control Register. When enabled, the PCSAMPLE scan register includes a data address sample. All load and store addresses can be captured, or they can be qualified using a data breakpoint trigger. DASQ=1 configures data sampling to record a data address only when it triggers data breakpoint 0. To be used for Data Address Sampling qualification, data breakpoint 0 must be enabled using its TE (trigger enable) bit.

PCSR controls how often data addresses are sampled. When the PCSR counter triggers, the most recent load/store address generated is accepted and made available to shift out through PCSAMPLE.

## 8.10 Fast Debug Channel

The Fast Debug Channel (FDC) mechanism provides an efficient means to transfer data between the M14K and an external device using the EJTAG TAP pins. The external device would typically be an EJTAG probe and that is the term used here, but it could be something else. FDC utilizes two First In First Out (FIFO) structures to buffer data between the M14K and probe. The probe uses the FDC TAP instruction to access these FIFOs, while the M14K itself accesses them using memory accesses. To transfer data out of the M14K, the M14K writes one or more pieces of data to the transmit FIFO. At this time, the M14K can resume doing other work. An external probe would examine the status of the transmit FIFO periodically. If there is data to be read, the probe starts to receive data from the FIFO, one entry at a time. When all data from the FIFO has been drained, the probe goes back to waiting for more data. The M14K can either choose to be informed of the empty transmit FIFO via an interrupt, or it can choose to periodically check the status. Receiving data works in a similar manner - the probe writes to the receive FIFO. At that time, the M14K is either interrupted, or finds out via polling a status bit. The M14K can then do load accesses to the receive FIFO and receive data being sent to it by the probe. The TAP transfer is bidirectional - a single shift can be pulling transmit data and putting receive data at the same time.

The primary advantage of FDC over normal processor accesses or fastdata accesses is that it does not require the M14K to be blocked when the probe is reading or writing to the data transfer FIFOs. This significantly reduces the M14K overhead and makes the data transfer far less intrusive to the code executing on the M14K.

Refer to the EJTAG Specification [11] for the general details on FDC. The remainder of this section describes implementation specific behavior and register values.

The FDC memory mapped registers are located in the common device memory map (CDMM) region. FDC has a device ID of 0xFD.



### 8.10.1 Common Device Memory Map

Software on the M14K accesses FDC through memory-mapped registers, located within the Common Device Memory Map (CDMM). The CDMM is a region of physical address space that is reserved for mapping IO device configuration registers within a MIPS processor. The base address and enabling of this region is controlled by the *CDMMBase* CP0 register, as described in [5.2.19 “CDMMBase Register \(CP0 Register 15, Select 2\)” on page 123](#).

Refer to Volume III of the MIPS Architecture Reference Manuals[11] for full details on the CDMM.

### 8.10.2 Fast Debug Channel Interrupt

The FDC block can generate an interrupt to inform software of incoming data being available or space being available in the outgoing FIFO. This interrupt is handled similarly to the timer or performance counter interrupts. The *CauseFDCI* bit indicates that the interrupt is pending. The interrupt is also sent to the core output *SI\_FDCI* where it is combined with one of the *SI\_Int* pins. For non-EIC mode, the *SI\_IPFDCI* input indicates which interrupt pin is has been combined with and this information is reflected in the *IntCtlIPFDCI* field. Note that this interrupt is a regular interrupt and not a debug interrupt.

The FDC Configuration Register (see [Section 8.10.6.2 “FDC Configuration \(FDCFG\) Register \(Offset 0x8\)”](#)) includes fields for enabling and setting the threshold for generating each interrupt. Receive and transmit interrupt thresholds are specified independently, but they are ORED together to form a single interrupt .

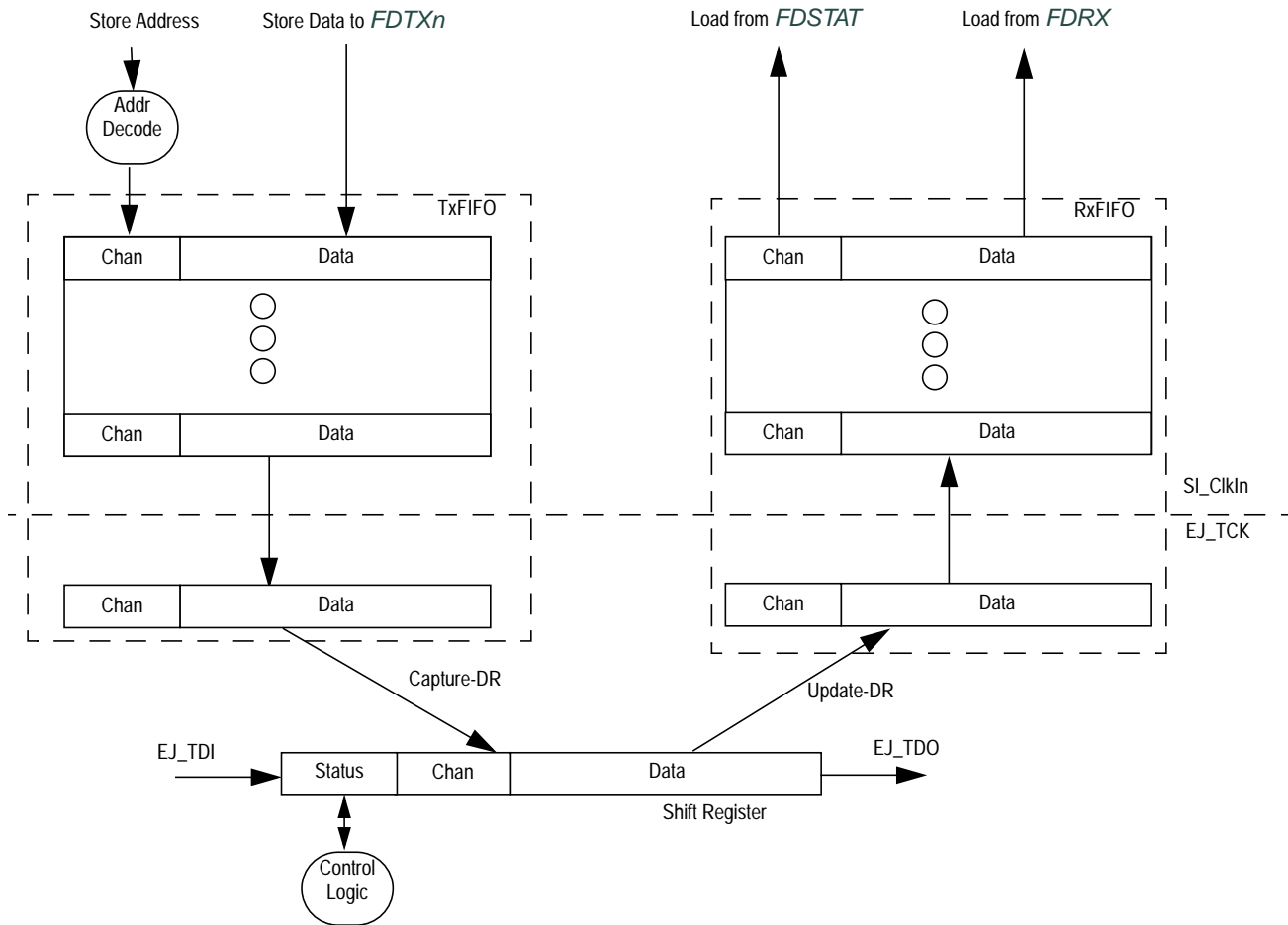
The following interrupt thresholds are supported:

- **Interrupts Disabled:** No interrupt will be generated and software must poll the status registers to determine if incoming data is available or if there is space for outgoing data.
- **Minimum M14K Overhead:** This setting minimizes the M14K overhead by not generating an interrupt until the receive FIFO (RxFIFO) is completely full or the transmit FIFO (TxFIFO) is completely empty.
- **Minimum latency:** To have the M14K take data as soon as it is available, the receive interrupt can be fired whenever the RxFIFO is not empty. There is a complimentary TxFIFO not full setting although that may not be quite as useful.
- **Maximum bandwidth:** When configured for minimum M14K overhead, bandwidth between the probe and M14K can be wasted if the M14K does not service the interrupt before the next transfer occurs. To reduce the chances of this happening, the interrupt threshold can be set to almost full or almost empty to generate an interrupt earlier. This setting causes receive interrupts to be generated when there are 0 or 1 unused RxFIFO entries. Transmit interrupts are generated when there are 0 or 1 used TxFIFO entries (see note in following section about this condition)

### 8.10.3 M14K™M14K FDC Buffers

[Figure 8.36](#) shows the general organization of the transmit and receive buffers on the M14K M14K.

Figure 8.36 Fast Debug Channel Buffer Organization



One particular thing to note is the asynchronous crossings between the *EJ\_TCK* and *SI\_ClkIn* clock domains. This crossing is handled with a handshake interface that safely transfers data between the domains. Two data registers are included in this interface, one in the source domain and one in the destination domain. The control logic actively manages these registers so that they can be used as FIFO entries. The fact that one FIFO entry is in the *EJ\_TCK* clock domain is normally transparent, but it can create some unexpected behavior:

- **TxFIFO availability:** Data is first written into the *SI\_ClkIn* FIFO entries, then into the *EJ\_TCK* FIFO entry, requiring several *EJ\_TCK* cycles to complete the handshake and move the data. *EJ\_TCK* is generally much slower than *SI\_ClkIn*, and may even be stopped (although that would be uncommon when this feature is in use). This can result in not enough space for new data, even though there are only N-1 data values queued up. To prevent the loss of data, the *TxF* flag in *FDSTAT* is set when all of the *SI\_ClkIn* FIFO entries are full. Software writes to the FIFO should always check the *TxF* bit before attempting the write and should not make any assumptions about being able to use all entries arbitrarily. i.e., software seeing the *FxE* bit set should not assume that it can write *TxCnt* data words without checking for full.
- **TxFIFO Almost Empty Interrupt:** As transmit data moves from *SI\_ClkIn* to *EJ\_TCK*, both of the flops will temporarily look full. This makes it difficult to determine when just 1 FIFO entry is in use. To enable a simpler condition, the almost empty TxInterrupt condition is set when all of the *SI\_ClkIn* FIFO entries are empty. When this

condition is met, there will be 0 or 1 valid entries. However, the interrupt will not be asserted when there is only one valid entry if it is an *SL\_ClkIn* entry

- The RxFIFO has similar characteristics, but these are even less visible to software since *SL\_ClkIn* must be running to access the FDC registers.

#### 8.10.4 Sleep mode

FDC data transfers do not prevent the core from entering sleep mode and will proceed normally in sleep mode. The FDC block monitors the TAP interface signals with a free-running clock. When new receive data is available or transmit data can be sent, the gated clock will be enabled for a few cycles to transfer the data and then allowed to stop again. If FDC interrupts are enabled, transferring data may cause an interrupt to be generated which can wake the core up.

#### 8.10.5 FDC TAP Register

The FDC TAP instruction performs a 38-bit bidirectional transfer of the FDC TAP register. The register format is shown in Figure 8.37 and the fields are described in Figure 8.42

**Figure 8.37 FDC TAP Register Format**

	37	36	35	32	31	0
In	Probe Data Accept	Data In Valid	ChannelID	Data		
Out	Receive Buffer Full	Data Out Valid				

**Table 8.42 FDC TAP Register Field Descriptions**

Fields		Description	Read / Write	Reset State
Name	Bits			
Probe Data Accept	37	Indicates to core that the probe is accepting the data that was scanned out.	W	Undefined
Data In Valid	36	Indicates to core that the probe is sending new data to the receive FIFO.	W	Undefined
Receive Buffer Full	37	Indicates to probe that the receive buffer is full and the core will not accept the data being scanned in. Analogous to ProbeDataAccept, but opposite polarity	R	0x0
Data Out Valid	36	Indicates to probe that the core is sending new data from the transmit FIFO	R	0
ChannelID	35:32	Channel number associated with the data being scanned in or out. This field can be used to indicate the type of data that is being sent and allow independent communication channels  Scanning in a value with ChannelID=0xd and Data In Valid = 0 will generate a receive interrupt. This can be used when the probe has completed sending data to the core.	R/W	Undefined

**Table 8.42 FDC TAP Register Field Descriptions**

Fields		Description	Read / Write	Reset State
Name	Bits			
Data	31:0	Data value being scanned in or out	R/W	Undefined

### 8.10.6 Fast Debug Channel Registers

This section describes the Fast Debug Channel registers. CPU access to FDC is via loads and stores to the FDC device in the Common Device Memory Map (CDMM) region. These registers provide access control, configuration and status information, as well as access to the transmit and receive FIFOs. The registers and their respective offsets are shown in [Table 8.43](#)

**Table 8.43 FDC Register Mapping**

Offset in CDMM device block	Register Mnemonic	Register Name and Description
0x0	FDACSR	FDC Access Control and Status Register
0x8	FDCFG	FDC Configuration Register
0x10	FDSTAT	FDC Status Register
0x18	FDRX	FDC Receive Register
0x20 + 0x8* n	FDTXn	FDC Transmit Register n (0 ≤ n ≤ 15)

#### 8.10.6.1 FDC Access Control and Status (FDACSR) Register (Offset 0x0)

This is the general CDMM Access Control and Status register which defines the device type and size and controls user and supervisor access to the remaining FDC registers. The Access Control and Status register itself is only accessible in kernel mode. [Figure 8.38](#) has the format of an Access Control and Status register (shown as a 64-bit register), and [Table 8.44](#) describes the register fields.

**Figure 8.38 FDC Access Control and Status Register**

63	32	31	24	23	22	21	16	15	12	11	4	3	2	1	0
0	DevID				0	DevSize				DevRev	0	Uw	Ur	Sw	Sr

**Table 8.44 FDC Access Control and Status Register Field Descriptions**

Fields		Description	Read / Write	Reset State
Name	Bits			
DevType	31:24	This field specifies the type of device.	R	0xfd
DevSize	21:16	This field specifies the number of extra 64-byte blocks allocated to this device. The value 0x2 indicates that this device uses 2 extra, or 3 total blocks.	R	0x2
DevRev	15:12	This field specifies the revision number of the device. The value 0x0 indicates that this is the initial version of FDC	R	0x0

**Table 8.44 FDC Access Control and Status Register Field Descriptions (Continued)**

Fields		Description	Read / Write	Reset State
Name	Bits			
Uw	3	This bit indicates if user-mode write access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to write to the device while in user mode with access disabled is ignored.	R/W	0
Ur	2	This bit indicates if user-mode read access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to read from the device while in user mode with access disabled will return 0 and not change any state.	R/W	0
Sw	1	This bit indicates if supervisor-mode write access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to write to the device while in supervisor mode with access disabled is ignored.	R/W	0
Sr	0	This bit indicates if supervisor-mode read access to this device is enabled. A value of 1 indicates that access is enabled. A value of 0 indicates that access is disabled. An attempt to read from the device while in supervisor mode with access disabled will return 0 and not change any state.	R/W	0
0	11:4	Reserved for future use. Ignored on write; returns zero on read.	R	0

**8.10.6.2 FDC Configuration (FDCFG) Register (Offset 0x8)**

The FDC configuration register holds information about the current configuration of the Fast Debug Channel mechanism. [Figure 8.39](#) has the format of the FDC Configuration register, and [Table 8.45](#) describes the register fields.

**Figure 8.39 FDC Configuration Register**

31	20	19	18	17	16	15	8	7	0
0	Tx_IntThresh		Rx_IntThresh		TxFIFOSize			RxFIFOSize	

**Table 8.45 FDC Configuration Register Field Descriptions**

Fields		Description	Read / Write	Reset State
Name	Bits			
0	31:20	Reserved for future use. Read as zeros, must be written as zeros.	R	0

Table 8.45 FDC Configuration Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State										
Name	Bits													
TxIntThresh	19:18	Controls whether transmit interrupts are enabled and the state of the TxFIFO needed to generate an interrupt.	R/W	0										
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Transmit Interrupt Disabled</td></tr><tr><td>1</td><td>Empty</td></tr><tr><td>2</td><td>Not Full</td></tr><tr><td>3</td><td>Almost Empty - zero or one entry in use*(see 8.10.2 for specifics)</td></tr></table>			Encoding	Meaning	0	Transmit Interrupt Disabled	1	Empty	2	Not Full	3	Almost Empty - zero or one entry in use*(see 8.10.2 for specifics)
		Encoding			Meaning									
		0			Transmit Interrupt Disabled									
		1			Empty									
		2			Not Full									
3	Almost Empty - zero or one entry in use*(see 8.10.2 for specifics)													
RxIntThresh	17:16	Controls whether receive interrupts are enabled and the state of the RxFIFO needed to generate an interrupt.	R/W	0										
		<table><tr><th>Encoding</th><th>Meaning</th></tr><tr><td>0</td><td>Receive Interrupt Disabled</td></tr><tr><td>1</td><td>Full</td></tr><tr><td>2</td><td>Not empty</td></tr><tr><td>3</td><td>Almost Full - zero or one entry free</td></tr></table>			Encoding	Meaning	0	Receive Interrupt Disabled	1	Full	2	Not empty	3	Almost Full - zero or one entry free
		Encoding			Meaning									
		0			Receive Interrupt Disabled									
		1			Full									
		2			Not empty									
3	Almost Full - zero or one entry free													
TxFIFOSize	15:8	This field holds the total number of entries in the transmit FIFO.	R	Preset										
RxFIFOSize	7:0	This field holds the total number of entries in the receive FIFO.	R	Preset										

### 8.10.6.3 FDC Status (FDSTAT) Register (Offset 0x10)

The FDC Status register holds up to date state information for the FDC mechanism. [Figure 8.40](#) shows the format of the FDC Status register, and [Table 8.46](#) describes the register fields.

Figure 8.40 FDC Status Register

31	24	23	16	15	8	7	4	3	2	1	0
Tx_Count		Rx_Count			0		RxChan	RxE	RxF	TxE	TxF

Table 8.46 FDC Status Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
Tx_Count	31:24	This optional field is not implemented and will read as 0	R	0
Rx_Count	23:16	This optional field is not implemented and will read as 0	R	0
0	15:8	Reserved for future use. Must be written as zeros and read as zeros.	R	0
RxChan	7:4	This field indicates the channel number used by the top item in the receive FIFO. This field is only valid if RxE=0.	R	Undefined

Table 8.46 FDC Status Register Field Descriptions (Continued)

Fields		Description	Read / Write	Reset State
Name	Bits			
RxE	3	If RxE is set, the receive FIFO is empty. If RxE is not set, the FIFO is not empty.	R	1
RxF	2	If RxF is set, the receive FIFO is full. If RxF is not set, the FIFO is not full.	R	0
TxE	1	If TxE is set, the transmit FIFO is empty. If TxE is not set, the FIFO is not empty.	R	1
TxF	0	If TxF is set, the transmit FIFO is full. If TxF is not set, the FIFO is not full.	R	0

#### 8.10.6.4 FDC Receive (FDRX) Register (Offset 0x18)

This register exposes the top entry in the receive FIFO. A read from this register returns the top item in the FIFO and removes it from the FIFO itself. The result of a write to this register is **UNDEFINED**. The result of a read when the FIFO is empty is also **UNDEFINED** so software must check the *RxE* flag in *FDSTAT* prior to reading. [Figure 8.41](#) shows the format of the *FDC Receive* register, and [Table 8.47](#) describes the register fields.

Figure 8.41 FDC Receive Register

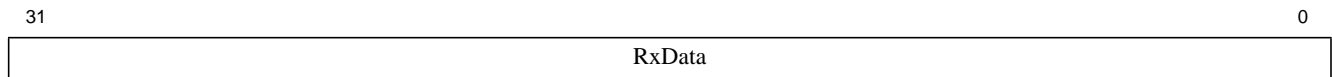


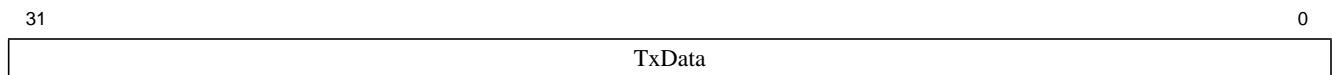
Table 8.47 FDC Receive Register Field Descriptions

Fields		Description	Read / Write	Reset State
Name	Bits			
RxData	31:0	This register holds the top entry in the receive FIFO	R	Undefined

#### 8.10.6.5 FDC Transmit n (FDTXn) Registers (Offset 0x20 + 0x8\*n)

These sixteen registers access the bottom entry in the transmit FIFO. The different addresses are used to generate a 4b channel identifier that is attached to the data value. This allows software to track different event types without needing to reserve a portion of the 32b data as a tag. A write to one of these registers results in a write to the transmit FIFO of the data value and channel ID corresponding to the register being written. Reads from these registers are **UNDEFINED**. Attempting to write to the transmit FIFO if it is full has **UNDEFINED** results. Hence, the software running on the core must check the *TxF* flag in *FDSTAT* to ensure that there is space for the write. [Figure 8.42](#) shows the format of the FDC Transmit register, and [Table 8.48](#) describes the register fields.

Figure 8.42 FDC Transmit Register



**Table 8.48 FDC Transmit Register Field Descriptions**

Fields		Description	Read / Write	Reset State
Name	Bits			
TxData	31:0	This register holds the bottom entry in the transmit FIFO	W, Undefined value on read	Undefined

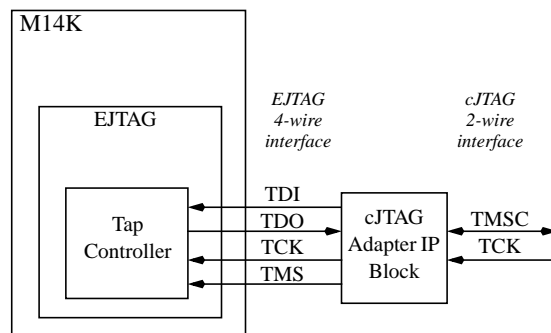
**Table 8.49 FDTXn Address Decode**

Address	Channel	Address	Channel	Address	Channel	Address	Channel
0x20	0x0	0x40	0x4	0x60	0x8	0x80	0xc
0x28	0x1	0x48	0x5	0x68	0x9	0x88	0xd
0x30	0x2	0x50	0x6	0x70	0xa	0x90	0xe
0x38	0x3	0x58	0x7	0x78	0xb	0x98	0xf

## 8.11 cJTAG Interface

The cJTAG external IP block, provided as part of the M14K processor core, converts a 4-wire EJTAG (IEEE 1149.1) interface to a 2-wire cJTAG (IEEE1149.7) interface. A high-level view of cJTAG is shown in [Figure 8.43](#). Operation of the conversion adapter is transparent to software.

Refer to the *cJTAG Adapter User's Manual* [12] for more details.

**Figure 8.43 cJTAG Interface**



## Instruction Set Overview

This chapter provides a general overview on the three CPU instruction set formats of the MIPS architecture: Immediate, Jump, and Register. Refer to [Chapter 10, “M14K™ Processor Core Instructions”](#) on page 239 for a complete listing and description of instructions.

This chapter discusses the following topics

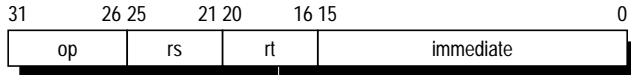
- [Section 9.1 “CPU Instruction Formats”](#)
- [Section 9.2 “Load and Store Instructions”](#)
- [Section 9.3 “Computational Instructions”](#)
- [Section 9.4 “Jump and Branch Instructions”](#)
- [Section 9.5 “Control Instructions”](#)
- [Section 9.6 “Coprocesor Instructions”](#)
- [Section 9.7 “Enhancements to the MIPS Architecture”](#)
- [Section 9.8 “MCU ASE Instructions”](#)

### 9.1 CPU Instruction Formats

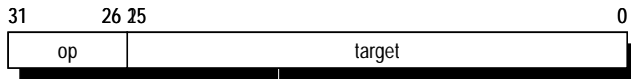
Each CPU instruction consists of a single 32-bit word, aligned on a word boundary. There are three instruction formats immediate (I-type), jump (J-type), and register (R-type)—as shown in [Figure 9.1](#). The use of a small number of instruction formats simplifies instruction decoding, allowing the compiler to synthesize more complicated (and less frequently used) operations and addressing modes from these three formats as needed.

Figure 9.1 Instruction Formats

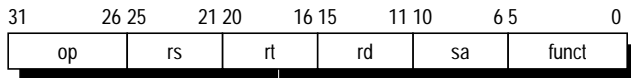
## I-Type (Immediate)



## J-Type (Jump)



## R-Type (Register)



op	6-bit operation code
rs	5-bit source register specifier
rt	5-bit target (source/destination) register or branch condition
immediate	16-bit immediate value, branch displacement or address displacement
target	26-bit jump target address
rd	5-bit destination register specifier
sa	5-bit shift amount
funct	6-bit function field

## 9.2 Load and Store Instructions

### 9.2.1 Scheduling a Load Delay Slot

A load instruction that does not allow its result to be used by the instruction immediately following is called a *delayed load instruction*. The instruction slot immediately following this delayed load instruction is referred to as the *load delay slot*.

In an M14K core, the instruction immediately following a load instruction can use the contents of the loaded register; however in such cases hardware interlocks insert additional real cycles. Although not required, the scheduling of load delay slots can be desirable, both for performance and R-Series processor compatibility.

### 9.2.2 Defining Access Types

*Access type* indicates the size of a core data item to be loaded or stored, set by the load or store instruction opcode.

Regardless of access type or byte ordering (endianness), the address given specifies the low-order byte in the addressed field. For a big-endian configuration, the low-order byte is the most-significant byte; for a little-endian configuration, the low-order byte is the least-significant byte.

The access type, together with the three low-order bits of the address, define the bytes accessed within the addressed word as shown in Table 9.1. Only the combinations shown in Table 9.1 are permissible; other combinations cause address error exceptions.

**Table 9.1 Byte Access Within a Word**

Access Type	Low Order Address Bits			Bytes Accessed							
				Big Endian (31-----0)				Little Endian (31-----0)			
	2	1	0	Byte				Byte			
Word	0	0	0	0	1	2	3	3	2	1	0
Triplebyte	0	0	0	0	1	2			2	1	0
	0	0	1		1	2	3	3	2	1	
Halfword	0	0	0	0	1					1	0
	0	1	0			2	3	3	2		
Byte	0	0	0	0							0
	0	0	1		1					1	
	0	1	0			2			2		
	0	1	1				3	3			

## 9.3 Computational Instructions

Computational instructions can be either in register (R-type) format, in which both operands are registers, or in immediate (I-type) format, in which one operand is a 16-bit immediate.

Computational instructions perform the following operations on register values:

- Arithmetic
- Logical
- Shift
- Multiply
- Divide

These operations fit in the following four categories of computational instructions:

- ALU Immediate instructions
- Three-operand Register-type Instructions
- Shift Instructions
- Multiply And Divide Instructions

### 9.3.1 Cycle Timing for Multiply and Divide Instructions

Any multiply instruction in the integer pipeline is transferred to the multiplier as remaining instructions continue through the pipeline; the product of the multiply instruction is saved in the HI and LO registers. If the multiply instruction is followed by an MFHI or MFLO before the product is available, the pipeline interlocks until this product does become available. Refer to [Chapter 2, “Pipeline of the M14K™ Core” on page 31](#) for more information on instruction latency and repeat rates.

## 9.4 Jump and Branch Instructions

Jump and branch instructions change the control flow of a program. All jump and branch instructions occur with a delay of one instruction: that is, the instruction immediately following the jump or branch (this is known as the instruction in the *delay slot*) always executes while the target instruction is being fetched from storage.

### 9.4.1 Overview of Jump Instructions

Subroutine calls in high-level languages are usually implemented with Jump or Jump and Link instructions, both of which are J-type instructions. In J-type format, the 26-bit target address shifts left 2 bits and combines with the high-order 4 bits of the current program counter to form an absolute address.

Returns, dispatches, and large cross-page jumps are usually implemented with the Jump Register or Jump and Link Register instructions. Both are R-type instructions that take the 32-bit byte address contained in one of the general purpose registers.

For more information about jump instructions, refer to the individual instructions in [Chapter 10, “M14K™ Processor Core Instructions” on page 239](#).

### 9.4.2 Overview of Branch Instructions

All branch instruction target addresses are computed by adding the address of the instruction in the delay slot to the 16-bit *offset* (shifted left 2 bits and sign-extended to 32 bits). All branches occur with a delay of one instruction.

If a conditional branch likely is not taken, the instruction in the delay slot is nullified.

Branches, jumps, ERET, and DERET instructions should not be placed in the delay slot of a branch or jump.

## 9.5 Control Instructions

Control instructions allow the software to initiate traps; they are always R-type.

## 9.6 Coprocessor Instructions

CP0 instructions perform operations on the System Control Coprocessor registers to manipulate the memory management and exception handling facilities of the processor. Refer to [Chapter 10, “M14K™ Processor Core Instructions” on page 239](#) for a listing of CP0 instructions.

## 9.7 Enhancements to the MIPS Architecture

The core execution unit implements the MIPS32 architecture, which includes the following instructions.

- CLOCount Leading Ones
- CLZCount Leading Zeros
- MADDMultiply and Add Word
- MADDUMultiply and Add Unsigned Word
- MSUBMultiply and Subtract Word
- MSUBUMultiply and Subtract Unsigned Word
- MULMultiply Word to Register
- SSNOPSuperscalar Inhibit NOP

### 9.7.1 CLO - Count Leading Ones

The CLO instruction counts the number of leading ones in a word. The 32-bit word in the GPR *rs* is scanned from most-significant to least-significant bit. The number of leading ones is counted and the result is written to the GPR *rd*. If all 32 bits are set in the GPR *rs*, the result written to the GPR *rd* is 32.

### 9.7.2 CLZ - Count Leading Zeros

The CLZ instruction counts the number of leading zeros in a word. The 32-bit word in the GPR *rs* is scanned from most-significant to least-significant bit. The number of leading zeros is counted and the result is written to the GPR *rd*. If all 32 bits are cleared in the GPR *rs*, the result written to the GPR *rd* is 32.

### 9.7.3 MADD - Multiply and Add Word

The MADD instruction multiplies two words and adds the result to the HI/LO register pair. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is added to the 64-bit concatenated values in the HI and LO register pair. The resulting value is then written back to the HI and LO registers. No arithmetic exception occurs under any circumstances.

### 9.7.4 MADDU - Multiply and Add Unsigned Word

The MADDU instruction multiplies two unsigned words and adds the result to the HI/LO register pair. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is added to the 64-bit concatenated values in the HI and LO register pair. The resulting value is then written back to the HI and LO registers. No arithmetic exception occurs under any conditions.

### 9.7.5 MSUB - Multiply and Subtract Word

The MSUB instruction multiplies two words and subtracts the result from the HI/LO register pair. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values in the HI and LO register pair. The resulting value is then written back to the HI and LO registers. No arithmetic exception occurs under any circumstances.

## 9.7.6 MSUBU - Multiply and Subtract Unsigned Word

The MSUBU instruction multiplies two unsigned words and subtracts the result from the HI/LO register pair. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as unsigned values, to produce a 64-bit result. The product is subtracted from the 64-bit concatenated values in the HI and LO register pair. The resulting value is then written back to the HI and LO registers. No arithmetic exception occurs under any circumstances.

## 9.7.7 MUL - Multiply Word

The MUL instruction multiplies two words and writes the result to a GPR. The 32-bit word value in the GPR *rs* is multiplied by the 32-bit value in the GPR *rt*, treating both operands as signed values, to produce a 64-bit result. The least-significant 32-bits of the product are written to the GPR *rd*. The contents of the HI and LO register pair are not defined after the operation. No arithmetic exception occurs under any circumstances.

## 9.7.8 SSNOP- Superscalar Inhibit NOP

The MIPS32 M14K processor cores treat this instruction as a regular NOP.

# 9.8 MCU ASE Instructions

The MCU ASE includes some new instructions which are particularly useful in microcontroller applications.

## 9.8.1 ACLR

This instruction allows a bit within an uncached I/O control register to be atomically cleared; that is, the read-modify byte write sequence performed by this instruction cannot be interrupted.

## 9.8.2 ASET

This instruction allows a bit within an uncached I/O control register to be atomically set; that is, the read-modify byte write sequence performed by this instruction cannot be interrupted.

## 9.8.3 IRET

This instruction can be used as a replacement for the ERET instruction when returning from an interrupt. This instruction implements the Automated Interrupt Epilogue feature, which automates restoring some of the COP0 registers from the stack and updating the CO\_Status register in preparation for returning to non-exception mode. This instruction also implements the optional Interrupt Chaining feature, which allows a subsequent interrupt to be handled without returning to non-exception mode.

## M14K™ Processor Core Instructions

This chapter supplements the MIPS32 Architecture Reference Manual by describing instruction behavior that is specific to a MIPS32 M14K processor core. The chapter is divided into the following sections:

- [Section 10.1 “Understanding the Instruction Descriptions”](#)
- [Section 10.2 “M14K™ Opcode Map”](#)
- [Section 10.3 “MIPS32® Instruction Set for the M14K™ core”](#)

The M14K processor core also supports the microMIPS extension to the MIPS32 architecture. The microMIPS instruction set is described in [Chapter 11, “microMIPS™ Instruction Set Architecture”](#) on page 275.

### 10.1 Understanding the Instruction Descriptions

Refer to Volume II of the MIPS32 Architecture Reference Manual for more information about the instruction descriptions. That document contains a description of the instruction fields, definition of terms, and functional notation.

### 10.2 M14K™ Opcode Map

#### Key

- CAPITALIZED text indicates an opcode mnemonic
- *Italicized* text indicates to look at the specified opcode submap for further instruction bit decode
- Entries containing the  $\alpha$  symbol indicate that a reserved instruction fault occurs if the core executes this instruction.
- Entries containing the  $\beta$  symbol indicate that a coprocessor unusable exception occurs if the core executes this instruction

**Table 10.1 Encoding of the Opcode Field**

opcode		bits 28..26							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	Special	RegImm	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	COP0	$\beta$	COP2	$\beta$	BEQL	BNEL	BLEZL	BGTZL
3	011	$\alpha$	$\alpha$	$\alpha$	$\alpha$	Special2	$\vartheta\Delta\Lambda\Xi$	$\alpha$	$\Sigma\pi\epsilon\chi\iota\alpha\lambda 3$
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	$\alpha$
5	101	SB	SH	SWL	SW	$\alpha$	$\alpha$	SWR	CACHE
6	110	LL	$\beta$	LWC2	PREF	$\alpha$	$\beta$	$\alpha$	$\alpha$
7	111	SC	$\beta$	SWC2	$\alpha$	$\alpha$	$\beta$	$\alpha$	$\alpha$

**Table 10.2 Special Opcode encoding of Function Field**

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL	$\beta$	SRL/ ROTR	SRA	SLLV	$\alpha$	SRLV/ ROTRV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	$\alpha$	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	$\alpha$	$\alpha$	$\alpha$	$\alpha$
3	011	MULT	MULTU	DIV	DIVU	$\alpha$	$\alpha$	$\alpha$	$\alpha$
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	$\alpha$	$\alpha$	SLT	SLTU	$\alpha$	$\alpha$	$\alpha$	$\alpha$
6	110	TGE	TGEU	TLT	TLTU	TEQ	$\alpha$	TNE	$\alpha$
7	111	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$

**Table 10.3 Special2 Opcode Encoding of Function Field**

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	$\alpha$	MSUB	MSUBU	$\alpha$	$\alpha$
1	001		$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
2	010	$UDI^1$ or $\alpha$							
3	011								
4	100	CLZ	CLO	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
5	101	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
6	110	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
7	111	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	SDBBP

1. CorExtend instructions are a build-time option of the M14K Pro core, if not implemented this instructions space will cause a reserved instruction exception. If assembler support exists, the mnemonics for CorExtend instructions are most likely UDI0, UDI1, ..., UDI15.



**Table 10.4 Special3 Opcode Encoding of Function Field**

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	EXT	$\alpha$	$\alpha$	$\alpha$	INS	$\alpha$	$\alpha$	$\alpha$
1	001	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
2	010	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
3	011	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
4	100	BSHFL	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
5	101	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
6	110	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
7	111	$\alpha$	$\alpha$	$\alpha$	PAHQP	$\alpha$	$\alpha$	$\alpha$	$\alpha$

**Table 10.5 RegImm Encoding of rt Field**

rt		bits 18..16							
		0	1	2	3	4	5	6	7
bits 20..19		000	001	010	011	100	101	110	111
0	00	BLTZ	BGEZ	BLTZL	BGEZL	$\alpha$	$\alpha$	$\alpha$	$\alpha$
1	01	TGEI	TGEIU	TLTI	TLTIU	TEQI	$\alpha$	TNEI	$\alpha$
2	10	BLTZAL	BGEZAL	BLTZALL	BGEZALL	$\alpha$	$\alpha$	$\alpha$	$\alpha$
3	11	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\Sigma\PNXI$

**Table 10.6 COP2 Encoding of rs Field**

rs		bits 23..21							
bits 25..24		0	1	2	3	4	5	6	7
		000	001	010	011	100	101	110	111
0	00	MFC2	$\alpha$	CFC2	MΦHX2	MTC2	$\alpha$	CTC2	MTHX2
1	01	BC2	$BC2^1$						
2	10	CO							
3	11								

1. The core will treat the entire row as a  $BC2$  instruction. However compiler and assembler support only exists for the first one. Some compiler and assembler products may allow the user to add new instructions.

**Table 10.7 COP2 Encoding of rt Field When rs=BC2**

rt		bits 16	
bits 17		0	1
0		BC2F	BC2T
1		BC2FL	BC2TL

**Table 10.8 COP0 Encoding of rs Field**

rs		bits 23..21							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC0	$\alpha$	$\alpha$	$\alpha$	MTC0	$\alpha$	$\alpha$	$\alpha$
1	01	$\alpha$	$\alpha$	РΔΠΓΠР	МΦМХ0	$\alpha$	$\alpha$	ΩРΠΓΠР	$\alpha$
2	10	CO							
3	11								

**Table 10.9 COP0 Encoding of Function Field When rs=CO**

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
1	001	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
2	010	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
3	011	ERET	IAXK	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	DERET
4	100	WAIT	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
5	101	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
6	110	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$
7	111	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$	$\alpha$

## 10.3 MIPS32® Instruction Set for the M14K™ core

This section describes the MIPS32 instructions for the M14K cores. [Table 10.10](#) lists the instructions in alphabetical order. Instructions that have implementation dependent behavior are described afterwards. The descriptions for other instructions exist in the architecture reference manual and are not duplicated here.

**Table 10.10 Instruction Set**

Instruction	Description	Function
ADD	Integer Add	$Rd = Rs + Rt$
ADDI	Integer Add Immediate	$Rt = Rs + \text{Immed}$
ADDIU	Unsigned Integer Add Immediate	$Rt = Rs +_{\text{U}} \text{Immed}$
ADDU	Unsigned Integer Add	$Rd = Rs +_{\text{U}} Rt$
AND	Logical AND	$Rd = Rs \& Rt$
ANDI	Logical AND Immediate	$Rt = Rs \& (0_{16} \parallel \text{Immed})$
ACLR	Atomic Bit Clear	See MCU ASE Instructions
ASET	Atomic Bit Set	See MCU ASE Instructions
B	Unconditional Branch (Assembler idiom for: BEQ r0, r0, offset)	$PC += (\text{int})\text{offset}$
BAL	Branch and Link (Assembler idiom for: BGEZAL r0, offset)	$GPR[31] = PC + 8$ $PC += (\text{int})\text{offset}$

**Table 10.10 Instruction Set (Continued)**

Instruction	Description	Function
BC2F	Branch On COP2 Condition False	if COP2Condition(cc) == 0 PC += (int)offset
BC2FL	Branch On COP2 Condition False Likely	if COP2Condition(cc) == 0 PC += (int)offset else Ignore Next Instruction
BC2T	Branch On COP2 Condition True	if COP2Condition(cc) == 1 PC += (int)offset
BC2TL	Branch On COP2 Condition True Likely	if COP2Condition(cc) == 1 PC += (int)offset else Ignore Next Instruction
BEQ	Branch On Equal	if Rs == Rt PC += (int)offset
BEQL	Branch On Equal Likely	if Rs == Rt PC += (int)offset else Ignore Next Instruction
BGEZ	Branch on Greater Than or Equal To Zero	if !Rs[31] PC += (int)offset
BGEZAL	Branch on Greater Than or Equal To Zero And Link	GPR[31] = PC + 8 if !Rs[31] PC += (int)offset
BGEZALL	Branch on Greater Than or Equal To Zero And Link Likely	GPR[31] = PC + 8 if !Rs[31] PC += (int)offset else Ignore Next Instruction
BGEZL	Branch on Greater Than or Equal To Zero Likely	if !Rs[31] PC += (int)offset else Ignore Next Instruction
BGTZ	Branch on Greater Than Zero	if !Rs[31] && Rs != 0 PC += (int)offset
BGTZL	Branch on Greater Than Zero Likely	if !Rs[31] && Rs != 0 PC += (int)offset else Ignore Next Instruction
BLEZ	Branch on Less Than or Equal to Zero	if Rs[31]    Rs == 0 PC += (int)offset
BLEZL	Branch on Less Than or Equal to Zero Likely	if Rs[31]    Rs == 0 PC += (int)offset else Ignore Next Instruction
BLTZ	Branch on Less Than Zero	if Rs[31] PC += (int)offset

Table 10.10 Instruction Set (Continued)

Instruction	Description	Function
BLTZAL	Branch on Less Than Zero And Link	GPR[31] = PC + 8 if Rs[31] PC += (int)offset
BLTZALL	Branch on Less Than Zero And Link Likely	GPR[31] = PC + 8 if Rs[31] PC += (int)offset else Ignore Next Instruction
BLTZL	Branch on Less Than Zero Likely	if Rs[31] PC += (int)offset else Ignore Next Instruction
BNE	Branch on Not Equal	if Rs != Rt PC += (int)offset
BNEL	Branch on Not Equal Likely	if Rs != Rt PC += (int)offset else Ignore Next Instruction
BREAK	Breakpoint	Break Exception
CACHE	Cache Operation	See Cache Description
CFC2	Move Control Word From Coprocessor 2	Rt = CCR[2, n]
CLO	Count Leading Ones	Rd = NumLeadingOnes(Rs)
CLZ	Count Leading Zeroes	Rd = NumLeadingZeroes(Rs)
COP0	Coprocessor 0 Operation	See Coprocessor Description
COP2	Coprocessor 2 Operation	See Coprocessor 2 Description
CTC2	Move Control Word To Coprocessor 2	CCR[2, n] = Rt
DERET	Return from Debug Exception	PC = DEPC Exit Debug Mode
DI	Disable Interrupts	Rt=Status Status <sub>IE</sub> =0
DIV	Divide	LO = (int)Rs / (int)Rt HI = (int)Rs % (int)Rt
DIVU	Unsigned Divide	LO = (uns)Rs / (uns)Rt HI = (uns)Rs % (uns)Rt
EHB	Execution Hazard Barrier	Stall until execution hazards are cleared
EI	Enable Interrupts	Rt=Status Status <sub>IE</sub> =1
ERET	Return from Exception	if SR[2] PC = ErrorEPC else PC = EPC SR[1] = 0 SR[2] = 0 LL = 0

Table 10.10 Instruction Set (Continued)

Instruction	Description	Function
EXT	Extract Bit Field	$Rt = \text{ExtractField}(Rs, msb, lsb)$
INS	Insert Bit Field	$Rt = \text{InsertField}(Rt, Rs, msb, lsb)$
IRET	Return from Exception	See MCU ASE Instructions
J	Unconditional Jump	$PC = PC[31:28] \parallel \text{offset} < 2$
JAL	Jump and Link	$GPR[31] = PC + 8$ $PC = PC[31:28] \parallel \text{offset} < 2$
JALR	Jump and Link Register	$Rd = PC + 8$ $PC = Rs$
JALR.HB	Jump and Link Register with Hazard Barrier	$Rd = PC + 8$ $PC = Rs$ Stall until all execution and instruction hazards are cleared
JR	Jump Register	$PC = Rs$
JR.HB	Jump Register with Hazard Barrier	$PC = Rs$ Stall until all execution and instruction hazards are cleared
LB	Load Byte	$Rt = (\text{byte})\text{Mem}[Rs + \text{offset}]$
LBU	Unsigned Load Byte	$Rt = (\text{ubyte})\text{Mem}[Rs + \text{offset}]$
LH	Load Halfword	$Rt = (\text{half})\text{Mem}[Rs + \text{offset}]$
LHU	Unsigned Load Halfword	$Rt = (\text{uhalf})\text{Mem}[Rs + \text{offset}]$
LL	Load Linked Word	$Rt = \text{Mem}[Rs + \text{offset}]$ $LL = 1$ $LL\text{Adr} = Rs + \text{offset}$
LUI	Load Upper Immediate	$Rt = \text{immediate} < 16$
LW	Load Word	$Rt = \text{Mem}[Rs + \text{offset}]$
LWC2	Load Word To Coprocessor 2	$CPR[2, n, 0] = \text{Mem}[Rs + \text{offset}]$
LWL	Load Word Left	See LWL instruction.
LWR	Load Word Right	See LWR instruction.
MADD	Multiply-Add	$HI, LO += (\text{int})Rs * (\text{int})Rt$
MFC0	Move From Coprocessor 0	$Rt = CPR[0, n, sel]$
MFC2	Move From Coprocessor 2	$Rt = CPR[2, n, sel_{31..0}]$
MFHC2	Move From High Word Coprocessor2	$Rt = CPR[2, n, sel]_{63..32}$
MFHI	Move From HI	$Rd = HI$
MFLO	Move From LO	$Rd = LO$
MOVN	Move Conditional on Not Zero	if $GPR[rt] \neq 0$ then $GPR[rd] = GPR[rs]$
MOVZ	Move Conditional on Zero	if $GPR[rt] = 0$ then $GPR[rd] = GPR[rs]$
MSUB	Multiply-Subtract	$HI, LO -= (\text{int})Rs * (\text{int})Rt$
MSUBU	Multiply-Subtract Unsigned	$HI, LO -= (\text{uns})Rs * (\text{uns})Rt$

Table 10.10 Instruction Set (Continued)

Instruction	Description	Function
MTC0	Move To Coprocessor 0	$CPR[0, n, sel] = Rt$
MTC2	Move To Coprocessor 2	$CPR[2, n, sel]_{31..0} = Rt$
MTHC2	Move To High Word Coprocessor 2	$CPR[2, n, sel]_{63..32} = Rt$
MTHI	Move To HI	$HI = Rs$
MTLO	Move To LO	$LO = Rs$
MUL	Multiply with register write	$HI \mid LO = \text{Unpredictable}$ $Rd = LO$
MULT	Integer Multiply	$HI \mid LO = (int)Rs * (int)Rd$
NOP	No Operation (Assembler idiom for: SLL r0, r0, r0)	
NOR	Logical NOR	$Rd = \sim(Rs \mid Rt)$
OR	Logical OR	$Rd = Rs \mid Rt$
ORI	Logical OR Immediate	$Rt = Rs \mid \text{Immed}$
PREF	Prefetch	Nop
RDHWR	Read HardWare Register	$Rt = HWR[Rd]$
RDPGPR	Read GPR from Previous Shadow Set	$Rd = SGPR[SRSCtl_{pss}, Rt]$
ROTR	Rotate Word Right	$Rd = Rt_{sa-1..0} \parallel Rt_{31..sa}$
ROTRV	Rotate Word Right Variable	$Rd = Rt_{Rs-1..0} \parallel Rt_{31..Rs}$
SB	Store Byte	$(byte)Mem[Rs+offset] = Rt$
SC	Store Conditional Word	if $LL = 1$ $mem[Rxoffsets] = Rt$ $Rt = LL$
SDBBP	Software Debug Breakpoint	Trap to SW Debug Handler
SEB	Sign Extend Byte	$Rd = \text{SignExtend}(Rt_{7..0})$
SEH	Sign Extend Half	$Rd = \text{SignExtend}(Rt_{15..0})$
SH	Store Halfword	$(half)Mem[Rs+offset] = Rt$
SLL	Shift Left Logical	$Rd = Rt \ll sa$
SLLV	Shift Left Logical Variable	$Rd = Rt \ll Rs[4:0]$
SLT	Set on Less Than	if $(int)Rs < (int)Rt$ $Rd = 1$ else $Rd = 0$
SLTI	Set on Less Than Immediate	if $(int)Rs < (int)Immed$ $Rt = 1$ else $Rt = 0$
SLTIU	Set on Less Than Immediate Unsigned	if $(uns)Rs < (uns)Immed$ $Rt = 1$ else $Rt = 0$

**Table 10.10 Instruction Set (Continued)**

Instruction	Description	Function
SLTU	Set on Less Than Unsigned	if (uns)Rs < (uns)Immed Rd = 1 else Rd = 0
SRA	Shift Right Arithmetic	Rd = (int)Rt >> sa
SRAV	Shift Right Arithmetic Variable	Rd = (int)Rt >> Rs[4:0]
SRL	Shift Right Logical	Rd = (uns)Rt >> sa
SRLV	Shift Right Logical Variable	Rd = (uns)Rt >> Rs[4:0]
SSNOP	Superscalar Inhibit No Operation	Nop
SUB	Integer Subtract	Rt = (int)Rs - (int)Rd
SUBU	Unsigned Subtract	Rt = (uns)Rs - (uns)Rd
SW	Store Word	Mem[Rs+offset] = Rt
SWC2	Store Word From Coprocessor 2	Mem[Rs+offset] = CPR[2, n, 0]
SWL	Store Word Left	See SWL instruction description.
SWR	Store Word Right	See SWR instruction description.
SYNC	Synchronize	See SYNC instruction below.
SYNCI	Synchronize Caches to Make Instruction Writes Effective	Nop
SYSCALL	System Call	SystemCallException
TEQ	Trap if Equal	if Rs == Rt TrapException
TEQI	Trap if Equal Immediate	if Rs == (int)Immed TrapException
TGE	Trap if Greater Than or Equal	if (int)Rs >= (int)Rt TrapException
TGEI	Trap if Greater Than or Equal Immediate	if (int)Rs >= (int)Immed TrapException
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	if (uns)Rs >= (uns)Immed TrapException
TGEU	Trap if Greater Than or Equal Unsigned	if (uns)Rs >= (uns)Rt TrapException
TLT	Trap if Less Than	if (int)Rs < (int)Rt TrapException
TLTI	Trap if Less Than Immediate	if (int)Rs < (int)Immed TrapException
TLTIU	Trap if Less Than Immediate Unsigned	if (uns)Rs < (uns)Immed TrapException
TLTU	Trap if Less Than Unsigned	if (uns)Rs < (uns)Rt TrapException
TNE	Trap if Not Equal	if Rs != Rt TrapException

**Table 10.10 Instruction Set (Continued)**

<b>Instruction</b>	<b>Description</b>	<b>Function</b>
TNEI	Trap if Not Equal Immediate	if Rs != (int)Immed TrapException
WAIT	Wait for Interrupts	Stall until interrupt occurs
WRPGPR	Write to GPR in Previous Shadow Set	SGPR[SRSCtl <sub>pss</sub> ,Rd]=Rt
WSBH	Word Swap Bytes within Halfwords	Rd=SwapBytesWithinHalfs(Rt)
XOR	Exclusive OR	Rd = Rs ^ Rt
XORI	Exclusive OR Immediate	Rt = Rs ^ (uns)Immed



31	26	25	21	20	16	15	14	12	11	4	3	0
REGIMM 000001	base					0	Bit	offset				
6	5					1	3	12				

**Format:** ACLR bit, offset(base)

**MIPS32 and MCU ASE**

**Purpose:** Atomically Clear Bit within Byte

**Description:** Disable interrupts;  $\text{temp} \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}]$ ;  $\text{temp} \leftarrow (\text{temp} \text{ and } \sim(1 \ll \text{bit}))$ ;  $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{temp}$ ; Enable Interrupts

The contents of the 8-bit byte at the memory location specified by the effective address are fetched. The specified bit within the byte is cleared to zero. The modified byte is stored in memory at the location specified by the effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address. The read-modify-write sequence cannot be interrupted.

Transactions with locking semantics occur in some memory interconnects/busses. It is implementation-specific whether this instruction uses such locking transactions.

#### Restrictions:

The operation of the processor is **UNDEFINED** if an ACLR instruction is executed in the delay slot of a branch or jump instruction.

#### Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian2)
TempIE ← StatusIE
StatusIE ← 0
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr1..0 xor BigEndianCPU2
temp ← memword7+8*byte..8*byte
temp ← temp and ((1 || 0bit) xor 0xFF)
dataword ← temp || 08*byte
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
StatusIE ← TempIE

```

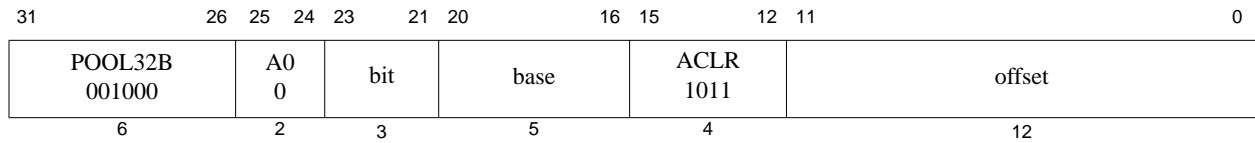
#### Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

#### Programming Notes:

Upon a TLB miss, a TLBS exception is signalled in the ExcCode field of the *Cause* register. For address error, a ADES exception is signalled in the ExcCode field of the *Cause* register. For other data-stream related exceptions such as Debug Data Break exceptions and Watch exceptions, it is implementation-specific whether this instruction is treated as a load or as a store.





**Format:** LR bit, offset(base)

**microMIPS and MCU ASE**

**Purpose:** Atomically Clear Bit within Byte

**Description:** Disable interrupts;  $\text{temp} \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}]$ ;  $\text{temp} \leftarrow (\text{temp} \text{ and } \sim(1 \ll \text{bit}))$ ;  $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{temp}$ ; Enable Interrupts

The contents of the byte at the memory location specified by the effective address are fetched. The specified bit within the byte is cleared to zero. The modified byte is stored in memory at the location specified by the effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address. The read-modify-write sequence cannot be interrupted. Transactions with locking semantics occur in some memory interconnects/busses. It is implementation-specific whether this instruction uses such locking transactions.

#### Restrictions:

The operation of the processor is **UNDEFINED** if an ACLR instruction is executed in the delay slot of a branch or jump instruction.

#### Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian2)
TempIE ← StatusIE
StatusIE ← 0
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr1..0 xor BigEndianCPU2
temp ← memword7+8*byte..8*byte
temp ← temp and ((1 || 0bit) xor 0xFF)
dataword ← temp || 08*byte
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
StatusIE ← TempIE

```

#### Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

#### Programming Notes:

Upon a TLB miss, a TLBS exception is signalled in the ExcCode field of the *Cause* register. For address error, a ADES exception is signalled in the ExcCode field of the *Cause* register. For other data-stream related exceptions such as Debug Data Break exceptions and Watch exceptions, it is implementation-specific whether this instruction is treated as a load or as a store.



31	26	25	21	20	16	15	14	12	11	4	3	0
REGIMM 000001						base		ATOMIC 00111		1	Bit	offset
6						5		5		1	3	12

**Format:** ASET bit, offset(base)

**MIPS32 and MCU ASE**

**Purpose:** Atomically Set Bit within Byte

**Description:** Disable interrupts;  $\text{temp} \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}]$ ;  $\text{temp} \leftarrow (\text{temp} \text{ or } (1 \ll \text{bit}))$ ;  $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{temp}$ ; Enable Interrupts

The contents of the 8-bit byte at the memory location specified by the effective address are fetched. The specified bit within the byte is set to one. The modified byte is stored in memory at the location specified by the effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address. The read-modify-write sequence cannot be interrupted.

Transactions with locking semantics occur in some memory interconnects/busses. It is implementation-specific whether this instruction uses such locking transactions.

#### Restrictions:

The operation of the processor is **UNDEFINED** if an ASET instruction is executed in the delay slot of a branch or jump instruction.

#### Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian2)
TempIE ← Status_IE
Status_IE ← 0
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_1..0 xor BigEndianCPU2
temp ← memword7+8*byte..8*byte
temp ← temp or ( 1 || 0bit )
dataword ← temp || 08*byte
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
Status_IE ← TempIE

```

#### Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

#### Programming Notes:

Upon a TLB miss, a TLBS exception is signalled in the ExcCode field of the *Cause* register. For address error, a ADES exception is signalled in the ExcCode field of the *Cause* register. For other data-stream related exceptions such as Debug Data Break exceptions and Watch exceptions, it is implementation-specific whether this instruction is treated as a load or as a store.



31	26	25	24	23	21	20	16	15	12	11	0
POOL32B 001000				A0 0	bit		base		ASET 0011		offset
6				2	3		5		4		12

**Format:** ASET bit, offset(base)

**microMIPS AND MCU ASE**

**Purpose:** Atomically Set Bit within Byte

**Description:** Disable interrupts;  $\text{temp} \leftarrow \text{memory}[\text{GPR}[\text{base}] + \text{offset}]$ ;  $\text{temp} \leftarrow (\text{temp} \text{ or } (1 \ll \text{bit}))$ ;  $\text{memory}[\text{GPR}[\text{base}] + \text{offset}] \leftarrow \text{temp}$ ; Enable Interrupts

The contents of the byte at the memory location specified by the effective address are fetched. The specified bit within the byte is set to one. The modified byte is stored in memory at the location specified by the effective address. The 12-bit signed *offset* is added to the contents of GPR *base* to form the effective address. The read-modify-write sequence cannot be interrupted.

Transactions with locking semantics occur in some memory interconnects/busses. It is implementation-specific whether this instruction uses such locking transactions.

#### Restrictions:

The operation of the processor is **UNDEFINED** if an ASET instruction is executed in the delay slot of a branch or jump instruction.

#### Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← pAddrPSIZE-1..2 || (pAddr1..0 xor ReverseEndian2)
TempIE ← StatusIE
StatusIE ← 0
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr1..0 xor BigEndianCPU2
temp ← memword7+8*byte..8*byte
temp ← temp or ( 1 || 0bit )
dataword ← temp || 08*byte
StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)
StatusIE ← TempIE

```

#### Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

#### Programming Notes:

Upon a TLB miss, a TLBS exception is signalled in the ExcCode field of the *Cause* register. For address error, a ADES exception is signalled in the ExcCode field of the *Cause* register. For other data-stream related exceptions such as Debug Data Break exceptions and Watch exceptions, it is implementation-specific whether this instruction is treated as a load or as a store.





31	26	25		6	5	0
COP0 010000	C0 1		0 00 0000 0000 0000 0000		IRET 111000	
6	1		20		6	

**Format:** IRET

**MIPS32 and MCU ASE**

**Purpose:** Interrupt Return with automated interrupt epilogue handling.

Optionally jump directly to another interrupt vector without returning to original return address.

### Description:

IRET is used to automate some of the operations that are required when returning from an interrupt handler. It can be used in place of the ERET instruction at the end of interrupt handlers. The IRET instruction is only appropriate when using Shadow Register Sets and EIC Interrupt mode. The automated operations of this instruction can be used to reverse the effects of the automated operations of the Auto-Prologue feature.

If the EIC mode of interrupts and the Interrupt Chaining feature are used, the IRET instruction can be used to shorten the time between returning from the current interrupt handler and handling the next requested interrupt.

If Automated Prologue feature is disabled, then IRET behaves exactly as ERET.

If either *Status*<sub>ERL</sub> or *Status*<sub>BEV</sub> bits are set, then IRET behaves exactly as ERET.

If Interrupt Chaining is disabled:

Interrupts are disabled. COP0 *Status*, *SRSCtl* and *EPC* registers are restored from the stack. GPR 29 is incremented for the stack frame size. IRET then clears execution and instruction hazards, conditionally restores *SRSCtl*<sub>CSS</sub> from *SRSCtl*<sub>PSS</sub>, and returns to the interrupted instruction pointed by the *EPC* register at the completion of interrupt processing.

If Interrupt Chaining is enabled:

Interrupts are disabled. COP0 *Status* register is restored from the stack. The priority output of the External Interrupt Controller is compared with the *IPL* field of the *Status* register.

If *Status*<sub>IPL</sub> has a higher priority than that of the External Interrupt Controller value:

COP0 *SRSCtl* and *EPC* registers are restored from the stack. GPR 29 is incremented for the stack frame size. IRET then clears execution and instruction hazards, conditionally restores *SRSCtl*<sub>CSS</sub> from *SRSCtl*<sub>PSS</sub>, and returns to the interrupted instruction pointed by the *EPC* register at the completion of interrupt processing.

If *Status*<sub>IPL</sub> field has a lower priority than that of the External Interrupt Controller value:

The value of GPR 29 is first saved to a temporary register then GPR 29 is incremented for the stack frame size. The EIC is signalled that the next pending interrupt has been accepted. This signalling will update the *Cause*<sub>R IPL</sub> and *SRSCtl*<sub>EICSS</sub> fields from the EIC output values. The *SRSCtl*<sub>EICSS</sub> field is copied to the *SRSCtl*<sub>CSS</sub> field while the *Cause*<sub>R IPL</sub> field is copied to the *Status*<sub>IPL</sub> field. The saved temporary register is copied to the GPR 29 of the current SRS. The *KSU*, *ERL* and *EXL* fields of the *Status* register are optionally set to zero. No barrier for execution hazards nor instruction hazards is created. IRET finishes by jumping to the interrupt vector driven by the EIC.

IRET does not execute the next instruction (i.e., it has no delay slot).

## Restrictions:

The operation of the processor is **UNDEFINED** if an IRET is executed in the delay slot of a branch or jump instruction.

The operation of the processor is **UNDEFINED** if an IRET is executed when either Shadow Register Sets are not enabled or when EIC interrupt mode is not enabled.

An IRET placed between an LL and SC instruction will always cause the SC to fail.

The effective addresses used for the stack memory transactions must be naturally-aligned. If either of the two least-significant bits of the address is non-zero, an Address Error exception occurs.

IRET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the IRET returns.

In a Release 2 implementation, IRET does not restore  $SRSCtl_{CSS}$  from  $SRSCtl_{PSS}$  if  $Status_{BEV} = 1$ , or if  $Status_{ERL} = 1$  because any exception that sets  $Status_{ERL}$  to 1 (Reset, Soft Reset, NMI, or cache error) does not save  $SRSCtl_{CSS}$  in  $SRSCtl_{PSS}$ . If software sets  $Status_{ERL}$  to 1, it must be aware of the operation of an IRET that may be subsequently executed.

The stack memory transactions behave as individual LW operations with respect to exception reporting. BadVAddr would report the faulting address for unaligned access and the faulting word address for unprivileged access, TLB Refill and TLB Invalid exceptions. For TLB exceptions, the faulting word address would be reflected in the *Context*, and *EntryHi* registers. The *CacheError* register would reflect the faulting word address for Cache Errors.

## Operation:

```
if (( IntCtlAPE == 0 ) | ( StatusERL == 1 ) | ( StatusBEV == 1 ))
    Act as ERET // read Operation section of ERET description
else
    if (ISAMode)
        EPC ← PC31..1 || 1 // in case of memory exception
    else
        EPC ← PC // in case of memory exception
    endif
    temp ← 0x4 + GPR[29]
    tempStatus ← LoadStackWord(temp)
    ClearHazards()
    if ( (IntCtlICE == 0) | ((IntCtlICE == 1) &
        (tempStatusIPL > EICRIPL)) )
        temp ← 0x8 + GPR[29]
        tempSRCtl ← LoadStackWord(temp)
        temp ← 0x0 + GPR[29]
        tempEPC ← LoadStackWord(temp)
    endif
    Status ← tempStatus
    if ( (IntCtlICE == 0) | ((IntCtlICE == 1) &
        (tempStatusIPL > EICRIPL)) )
        GPR[29] ← GPR[29] + DecodedValue(IntCtlStkDec)
        SRSCtl ← tempSRCtl
        EPC ← tempEPC
        temp ← EPC
        StatusEXL ← 0
        if (ArchitectureRevision ≥ 2) and (SRSCtlHSS > 0)
```

```

    and (StatusBEV = 0) then
        SRSCtlCSS ← SRSCtlPSS
    endif
    if IsMicroMIPSImplemented() then
        PC ← temp31..1 || 0
        ISAMode ← temp0
    else
        PC ← temp
    endif
    LLbit ← 0
    CauseIC ← 0
    ClearHazards()
else
    Signal_EIC_for_Next_Interrupt()
    (wait for EIC outputs to update)
    CauseRIPL ← EICRIPL
    SRSCtlEICSS ← EICSS
    temp29 ← GPR[29]
    GPR[29] ← GPR[29] + DecodedValue(IntCtlStkDec)
    StatusIPL ← CauseRIPL
    SRSCtlCSS ← SRSCtlEICSS
    NewShadowSet ← SRSCtlEICSS
    GPR[29] ← temp29
    if (IntCtlClrEXL == 1)
        StatusEXL ← 0
        StatusKSU ← 0
    endif
    CauseIC ← 1
    ClearHazards()
    PC ← CalcIntrptAddress()
endif
endif

function LoadStackWord(vaddr)
    if vAddr1..0 ≠ 02 then
        SignalException(AddressError)
    endif
    (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
    memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
    LoadStackWord ← memword
endfunction LoadStackWord

function CalcIntrptAddress()
    if StatusBEV = 1
        vectorBase ← 0xBFC0.0200
    else
        if ( ArchitectureRevision ≥ 2)
            vectorBase ← EBase31..12 || 011
        else
            vectorBase ← 0x8000.0000
        endif
    endif
    if (CauseIV = 0)
        vectorOffset = 0x180
    else

```

```

if (StatusBEV = 1) or (IntCtlVS = 0)
    vectorOffset = 0x200
else
    if ( Config3VEIC = 1 and EIC_Option=1)
        VectorNum = CauseRIPL
    elseif (Config3VEIC = 1 and EIC_Option=2)
        VectorNum = EIC_VectorNum
    elseif (Config3VEIC = 0 )
        VectorNum = VIntPriorityEncoder()
    endif
    if (Config3VEIC = 1 and EIC_Option=3)
        vectorOffset = EIC_VectorOffset
    else
        vectorOffset = 0x200 + (VectorNum x (IntCtlVS || 05))
    endif
endif

endif
CalcIntrptAddress = vectorBase | vectorOffset
endfunction CalcIntrptAddress

```

#### Exceptions:

Coprocessor Unusable Exception, TLB Refill, TLB Invalid, Address Error, Watch, Cache Error, Bus Error  
Exceptions

31	26	25	6	5	0
POOL32A 000000	000 0000 0011 0100 1101				POOL32AXf 111100
6	20				6

**Format:** IRET

microMIPS and MCU ASE

**Purpose:** Interrupt Return with automated interrupt epilogue handling.

Optionally jump directly to another interrupt vector without returning to original return address.

### Description:

IRET automates some of the operations that are required when returning from an interrupt handler and can be used in place of the ERET instruction at the end of interrupt handlers. IRET is only appropriate when using Shadow Register Sets and the EIC Interrupt mode. The automated operations of this instruction can be used to reverse the effects of the automated operations of the Auto-Prologue feature.

If the EIC interrupt mode and the Interrupt Chaining feature are used, the IRET instruction can be used to shorten the time between returning from the current interrupt handler and handling the next requested interrupt.

If the Automated Prologue feature is disabled, then IRET behaves exactly like ERET.

If either the *Status*<sub>ERL</sub> or *Status*<sub>BEV</sub> bits are set, then IRET behaves exactly like ERET.

If Interrupt Chaining is disabled:

Interrupts are disabled. COP0 *Status*, *SRSCtl*, and *EPC* registers are restored from the stack. GPR 29 is incremented for the stack frame size. IRET then clears execution and instruction hazards, conditionally restores *SRSCtl*<sub>CSS</sub> from *SRSCtl*<sub>PSS</sub>, and returns at the completion of interrupt processing to the interrupted instruction pointed to by the *EPC* register. If Interrupt Chaining is enabled:

Interrupts are disabled. COP0 *Status* register is restored from the stack. The priority output of the External Interrupt Controller is compared with the IPL field of the *Status* register.

If *Status*<sub>IPL</sub> has a higher priority than the External Interrupt Controller value:

COP0 *SRSCtl* and *EPC* registers are restored from the stack. GPR 29 is incremented for the stack frame size. IRET then clears execution and instruction hazards, conditionally restores *SRSCtl*<sub>CSS</sub> from *SRSCtl*<sub>PSS</sub>, and returns to the interrupted instruction pointed to by the *EPC* register at the completion of interrupt processing.

If *Status*<sub>IPL</sub> has a lower priority than the External Interrupt Controller value:

The value of GPR 29 is first saved to a temporary register and then GPR 29 is incremented for the stack frame size. The EIC is signalled that the next pending interrupt has been accepted. This signalling will update the *Cause*<sub>R IPL</sub> and *SRSCtl*<sub>EICSS</sub> fields from the EIC output values. The *SRSCtl*<sub>EICSS</sub> field is copied to the *SRSCtl*<sub>CSS</sub> field, while the *Cause*<sub>R IPL</sub> field is copied to the *Status*<sub>IPL</sub> field. The saved temporary register is copied to the GPR 29 of the current SRS. The KSU and EXL fields of the *Status* register are optionally set to zero. No barrier for execution hazards or instruction hazards is created. IRET finishes by jumping to the interrupt vector driven by the EIC.

IRET does not execute the next instruction (i.e., it has no delay slot).

## Restrictions:

The operation of the processor is **UNDEFINED** if IRET is executed in the delay slot of a branch or jump instruction.

The operation of the processor is **UNDEFINED** if IRET is executed when either Shadow Register Sets are not enabled, or the EIC interrupt mode is not enabled.

An IRET placed between an LL and SC instruction will always cause the SC to fail.

The effective addresses used for stack transactions must be naturally-aligned. If either of the two least-significant bits of the address is non-zero, an Address Error exception occurs.

IRET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier begin with the instruction fetch and decode of the instruction at the PC to which the IRET returns.

In a Release 2 implementation, IRET does not restore  $SRSCtl_{CSS}$  from  $SRSCtl_{PSS}$  if  $Status_{BEV} = 1$  or  $Status_{ERL} = 1$ , because any exception that sets  $Status_{ERL}$  to 1 (Reset, Soft Reset, NMI, or cache error) does not save  $SRSCtl_{CSS}$  in  $SRSCtl_{PSS}$ . If software sets  $Status_{ERL}$  to 1, it must be aware of the operation of an IRET that may be subsequently executed.

The stack transactions behave as individual LW operations with respect to exception reporting. BadVAddr would report the faulting address for an unaligned access, and the faulting word address for unprivileged access, TLB Refill, and TLB Invalid exceptions. For TLB exceptions, the faulting word address would be reflected in the *Context* and *EntryHi* registers. The *CacheError* register would reflect the faulting word address for Cache Errors.

## Operation:

```
if (( IntCtlAPE == 0) | (StatusERL == 1) | (StatusBEV == 1))
    Act as ERET // read Operation section of ERET description
else
    if (ISAMode)
        EPC ← PC31..1 || 1 // in case of memory exception
    else
        EPC ← PC // in case of memory exception
    endif
    temp ← 0x4 + GPR[29]
    tempStatus ← LoadStackWord(temp)
    ClearHazards()
    if ( (IntCtlICE == 0) | ((IntCtlICE == 1) &
        (tempStatusIPL > EICRIPL)) )
        temp ← 0x8 + GPR[29]
        tempSRSCtl ← LoadStackWord(temp)
        temp ← 0x0 + GPR[29]
        tempEPC ← LoadStackWord(temp)
    endif
    Status ← tempStatus
    if ( (IntCtlICE == 0) | ((IntCtlICE == 1) &
        (tempStatusIPL > EICRIPL)) )
        GPR[29] ← GPR[29] + DecodedValue(IntCtlStkDec)
        SRSCtl ← tempSRSCtl
        EPC ← tempEPC
        temp ← EPC
        StatusEXL ← 0
        if (ArchitectureRevision ≥ 2) and (SRSCtlHSS > 0) and (StatusBEV = 0) then
            SRSCtlCSS ← SRSCtlPSS
        endif
        if IsMicroMIPSImplemented() then
```

```

        PC ← temp31..1 || 0
        ISAMode ← temp0
    else
        PC ← temp
    endif
    LLbit ← 0
    CauseIC ← 0
    ClearHazards()
else
    Signal_EIC_for_Next_Interrupt()
    (wait for EIC outputs to update)
    CauseRIPL ← EICRIPL
    SRSCtlEICSS ← EICSS
    temp29 ← GPR[29]
    GPR[29] ← GPR[29] + DecodedValue(IntCtlStkDec)
    StatusIPL ← CauseRIPL
    SRSCtlCSS ← SRSCtlEICSS
    NewShadowSet ← SRSCtlEICSS
    GPR[29] ← temp29
    if (IntCtlClrEXL == 1)
        StatusEXL ← 0
        StatusKSU ← 0
    endif
    CauseIC ← 1
    ClearHazards()
    PC ← CalcIntrptAddress()
endif
endif

function LoadStackWord(vaddr)
    if vAddr1..0 ≠ 02 then
        SignalException(AddressError)
    endif
    (pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
    memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
    LoadStackWord ← memword
endfunction LoadStackWord

function CalcIntrptAddress()
    if StatusBEV = 1
        vectorBase ← 0xBFC0.0200
    else
        if ( ArchitectureRevision ≥ 2)
            vectorBase ← EBase31..12 || 011
        else
            vectorBase ← 0x8000.0000
        endif
    endif
    if (CauseIV = 0)
        vectorOffset = 0x180
    else
        if (StatusBEV = 1) or (IntCtlVS = 0)
            vectorOffset = 0x200
        else
            if ( Config3VEIC = 1 and EIC_Option=1)

```

```

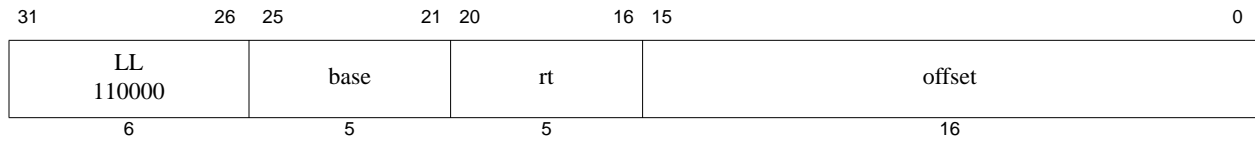
        VectorNum = CauseRIPL
    elseif (Config3VEIC = 1 and EIC_Option=2)
        VectorNum = EIC_VectorNum
    elseif (Config3VEIC = 0 )
        VectorNum = VIntPriorityEncoder()
    endif
    if (Config3VEIC = 1 and EIC_Option=3)
        vectorOffset = EIC_VectorOffset
    else
        vectorOffset = 0x200 + (VectorNum x (IntCtlVS || 05))
    endif
endif
endif
CalcIntrptAddress = vectorBase | vectorOffset
endfunction CalcIntrptAddress

```

**Exceptions:**

Coprocessor Unusable Exception, TLB Refill, TLB Invalid, Address Error, Watch, Cache Error, Bus Error  
Exceptions





**Format:** LL rt, offset(base)

**MIPS32**

**Purpose:** Load Linked Word

To load a word from memory for an atomic read-modify-write

**Description:**  $GPR[rt] \leftarrow memory[GPR[base] + offset]$

The LL and SC instructions provide the primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched and written into GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

This begins a RMW sequence on the current processor. There can be only one active RMW sequence per processor. When an LL is executed it starts an active RMW sequence replacing any other sequence that was active. The RMW sequence is completed by a subsequent SC instruction that either completes the RMW sequence atomically and succeeds, or does not and fails.

Executing LL on one processor does not cause an action that, by itself, causes an SC for the same block to fail on another processor.

An execution of LL does not have to be followed by execution of SC; a program is free to abandon the RMW sequence without attempting a write.

#### Restrictions:

The addressed location must be synchronizable by all processors and I/O devices sharing the location; if it is not, the result is **UNPREDICTABLE**. Which storage is synchronizable is a function of both CPU and system implementations. See the documentation of the SC instruction for the formal definition. The addressed location may be uncached for the M14K core.

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the effective address is non-zero, an Address Error exception occurs.

#### Operation:

```

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
memword ← LoadMemory(CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
LLbit ← 1

```

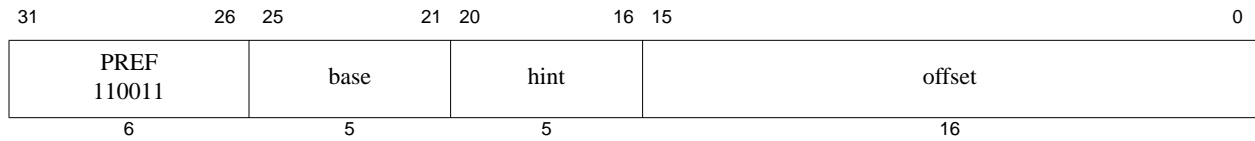
#### Exceptions:

TLB Refill, TLB Invalid, Address Error, Reserved Instruction, Watch

#### Programming Notes:

There is no Load Linked Word Unsigned operation corresponding to Load Word Unsigned.





**Format:** `PREF hint,offset(base)`

**MIPS32**

**Purpose:** Prefetch

To move data between memory and cache.

**Description:** `prefetch_memory(GPR[base] + offset)`

PREF adds the 16-bit signed *offset* to the contents of GPR *base* to form an effective byte address. The *hint* field supplies information about the way that the data is expected to be used.

PREF does not cause addressing-related exceptions, including TLB exceptions. If the address specified would cause an addressing exception, the exception condition is ignored and no data movement occurs. However even if no data is moved, some action that is not architecturally visible, such as writeback of a dirty cache line, can take place.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREF instruction.

PREF neither generates a memory operation nor modifies the state of a cache line for a location with an *uncached* memory access type, whether this type is specified by the address segment (e.g., *kseg1*), the programmed coherency attribute of a segment (e.g., the use of the *K0*, *KU*, or *K23* fields in the *Config* register), or the per-page coherency attribute provided by the TLB.

If PREF results in a memory operation, the memory access type and coherency attribute used for the operation are determined by the memory access type and coherency attribute of the effective address, just as it would be if the memory operation had been caused by a load or store to the effective address.

### Values of *hint* Field for PREF Instruction

Value	Name	Data Use and Desired Prefetch Action
0	load	Use: Prefetched data is expected to be read (not modified). Action: Fetch data as if for a load.
1	store	Use: Prefetched data is expected to be stored or modified. Action: Fetch data as if for a store.
2-3	Reserved	
4	load_streamed	Use: Prefetched data is expected to be read (not modified) but not reused extensively; it “streams” through cache.
5	store_streamed	Use: Prefetched data is expected to be stored or modified but not reused extensively; it “streams” through cache.
6	load_retained	Use: Prefetched data is expected to be read (not modified) and reused extensively; it should be “retained” in the cache.
7	store_retained	Use: Prefetched data is expected to be stored or modified and reused extensively; it should be “retained” in the cache.

### Values of *hint* Field for PREF Instruction

8-24	Reserved	
25	writeback_invalidate (also known as “nudge”)	
26-29		
30		
31		

#### Restrictions:

None

#### Operation:

```
vAddr ← GPR[base] + sign_extend(offset)
(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)
Prefetch(CCA, pAddr, vAddr, DATA, hint)
```

#### Exceptions:

Bus Error, Cache Error

Prefetch does not take any TLB-related or address-related exceptions under any circumstances.

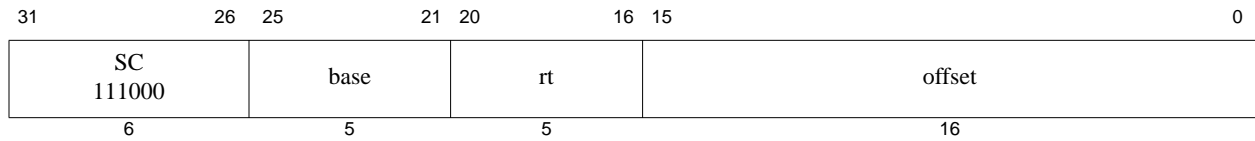
#### Programming Notes:

Prefetch cannot move data to or from a mapped location unless the translation for that location is present in the TLB. Locations in memory pages that have not been accessed recently may not have translations in the TLB, so prefetch may not be effective for such locations.

Prefetch does not cause addressing exceptions. A prefetch may be used using an address pointer before the validity of the pointer is determined without worrying about an addressing exception.

It is implementation dependent whether a Bus Error or Cache Error exception is reported if such an error is detected as a byproduct of the action taken by the PREF instruction. Typically, this only occurs in systems which have high-reliability requirements.

Prefetch operations have no effect on cache lines that were previously locked with the CACHE instruction.



**Format:** SC *rt*, *offset*(*base*)

**MIPS32**

**Purpose:** Store Conditional Word

To store a word to memory to complete an atomic read-modify-write

**Description:** if atomic\_update then memory[GPR[base] + offset]  $\leftarrow$  GPR[rt], GPR[rt]  $\leftarrow$  1  
else GPR[rt]  $\leftarrow$  0

The LL and SC instructions provide primitives to implement atomic read-modify-write (RMW) operations for synchronizable memory locations.

The 32-bit word in GPR *rt* is conditionally stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form an effective address.

The SC completes the RMW sequence begun by the preceding LL instruction executed on the processor. To complete the RMW sequence atomically, the following occur:

- The 32-bit word of GPR *rt* is stored into memory at the location specified by the aligned effective address.
- A 1, indicating success, is written into GPR *rt*.

Otherwise, memory is not modified and a 0, indicating failure, is written into GPR *rt*. On the M14K core, the SRAM interface supports a lock protocol and the success or failure can be indicated by external hardware.

If the following event occurs between the execution of LL and SC, the SC fails:

- An ERET instruction is executed.

If either of the following events occurs between the execution of LL and SC, the SC may succeed or it may fail; the success or failure is not predictable. Portable programs should not cause one of these events.

- A memory access instruction (load, store, or prefetch) is executed on the processor executing the LL/SC.
- The instructions executed starting with the LL and ending with the SC do not lie in a 2048-byte contiguous region of virtual memory. (The region does not have to be aligned, other than the alignment required for instruction words.)

The following conditions must be true or the result of the SC is **UNPREDICTABLE**:

- Execution of SC must have been preceded by execution of an LL instruction.
- An RMW sequence executed without intervening events that would cause the SC to fail must use the same address in the LL and SC. The address is the same if the virtual address, physical address, and cache-coherence algorithm are identical.

#### Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

### Operation:

```
vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
if LLbit then
    StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 031 || LLbit
```

### Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

### Programming Notes:

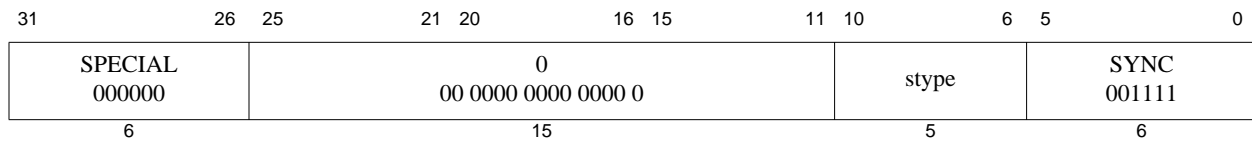
LL and SC are used to atomically update memory locations, as shown below.

```
L1:
    LL      T1, (T0) # load counter
    ADDI    T2, T1, 1 # increment
    SC      T2, (T0) # try to store, checking for atomicity
    BEQ     T2, 0, L1 # if not atomic (0), try again
    NOP                    # branch-delay slot
```

Exceptions between the LL and SC cause SC to fail, so persistent exceptions must be avoided. Some examples of these are arithmetic operations that trap, system calls, and floating point operations that trap or require software emulation assistance.

LL and SC function on a single processor for *cached noncoherent* memory so that parallel programs can be run on uniprocessor systems that do not support *cached coherent* memory access types.





**Format:** SYNC (stype = 0 implied)

**MIPS32**

**Purpose:** Synchronize Shared Memory

To order loads and stores.

**Description:**

*Simple Description:*

- SYNC affects only *uncached* and *cached coherent* loads and stores. The loads and stores that occur before the SYNC must be completed before the loads and stores after the SYNC are allowed to start.
- Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.
- SYNC is required, potentially in conjunction with SSNOP (in Release 1 of the Architecture) or EHB (in Release 2 of the Architecture), to guarantee that memory reference results are visible across operating mode changes. For example, a SYNC is required on entry to and exit from Debug Mode to guarantee that memory affects are handled correctly.

*Detailed Description:*

- SYNC does not guarantee the order in which instruction fetches are performed. The *stype* values 1-31 are reserved for future extensions to the architecture. A value of zero will always be defined such that it performs all defined synchronization operations. Non-zero values may be defined to remove some synchronization operations. As such, software should never use a non-zero value of the *stype* field, as this may inadvertently cause future failures if non-zero values remove synchronization operations.
- The SYNC instruction is externalized on the SRAM interface of the M14K core. External logic can use this information in a system-dependent manner to enforce memory ordering between various memory elements in the system.

**Restrictions:**

The effect of SYNC on the global order of loads and stores for memory access types other than *uncached* and *cached coherent* is **UNPREDICTABLE**.

**Operation:**

SyncOperation(stype)

**Exceptions:**

None



31	26	25	24		6	5	0
COP0 010000				CO 1	Implementation-Dependent Code		WAIT 100000
6				1	19		6

**Format:** WAIT

**MIPS32**

**Purpose:** Enter Standby Mode

Wait for Event

### Description:

The WAIT instruction forces the core into low power mode. The pipeline is stalled and when all external requests are completed, the processor's main clock is stopped. The processor will restart when reset (*SL\_Reset* or *SL\_ColdReset*) is signaled, or a non-masked interrupt is taken (*SI\_NMI*, *SI\_Int*, or *EJ\_DINT*). Note that the M14K core does not use the code field in this instruction.

If the pipeline restarts as the result of an enabled interrupt, that interrupt is taken between the WAIT instruction and the following instruction (EPC for the interrupt points at the instruction following the WAIT instruction).

### Restrictions:

The operation of the processor is **UNDEFINED** if a WAIT instruction is placed in the delay slot of a branch or a jump.

If access to Coprocessor 0 is not enabled, a Coprocessor Unusable Exception is signaled.

### Operation:

**I:** Enter lower power mode  
**I+1:** /\* Potential interrupt taken here \*/

### Exceptions:

Coprocessor Unusable Exception



## microMIPS™ Instruction Set Architecture

The microMIPS™ architecture minimizes the code footprint of applications, thus reducing the cost of memory, which is particularly high for embedded memory. At the same time, the high performance of MIPS cores is maintained. Using this technology, the customer can generate best results without spending time to profile its application. The smaller code footprint typically leads to reduced power consumption per executed task because of the smaller number of memory accesses.

microMIPS is a replacement for the existing MIPS16e ASE. It is also an alternative to the MIPS32 instruction encoding and can be implemented in parallel or stand-alone.

Overview of changes from the existing MIPS32ISA:

- 16-bit and 32-bit opcodes, for MIPS64 also 48-bit opcodes
- Optimized opcode/operand field definitions based on statistics
- Branch and jump delay slots will be kept for maximum compatibility and lowest risk
- Removal of branch likely instructions, emulation by assembler
- Fine-tuned register allocation algorithm in the compiler for lowest code size

### 11.1 Overview

#### 11.1.1 MIPSr3™ Architecture

MIPSr3™ is a family of architectures which includes Release 3.0 of the MIPS32 Architecture and the first release of the microMIPS32 architecture. Enhancements included in the MIPSr3 Architecture are:

- MIPSr3 Architecture includes MIPS32 Release 3 ISA and microMIPS ISA.
- The MIPS16 ASE is phased out; microMIPS is a replacement for MIPS16e. Therefore these two ASEs never co-exist within the same processor core.
- Branch likely instructions are phased out in microMIPS and are emulated by the assembler. They remain available in the MIPS32 encoding.

Unless otherwise described in this document, all other aspects of the MIPS32r3 architecture are identical to MIPS32 Release 2.

### 11.1.2 Default ISA Mode

The instruction sets that are available in an implementation are reported in the *Config3*<sub>ISA</sub> register field (bits 15:14). *Config1*<sub>CA</sub> (bit 2) is not used for microMIPS.

For implementations that support both microMIPS and MIPS32, the selected ISA mode following reset is determined by the setting of the *Config3*<sub>ISA</sub> register field., which is a read-only field set by a hardware signal external to the processor core.

For implementations that support both microMIPS and MIPS32, the selected ISA mode upon handling an exception is determined by the setting of the *Config3*<sub>ISAOnExc</sub> register field (bit 16). The *Config3*<sub>ISAOnExc</sub> register field is writeable by software and has a reset value that is set by a hardware signal external to the processor core. This register field allows privileged software to change the ISA mode to be used for subsequent exceptions. This capability is for all exception types whose vectors are offsets of the *EBASE* register.

For implementations that support both microMIPS and MIPS32, the selected ISA mode when handling a debug exception is determined by the setting of the ISAOnDebug register field in the *EJTAG TAP Control* register. This register field is writeable by EJTAG probe software and has a reset value that is set by a hardware signal external to the processor core.

For CPU cores supporting the MT ASE and multiple VPEs, the ISA mode for exceptions can be selected on a per-VPE basis.

### 11.1.3 Software Detection

Software can determine if microMIPS is implemented by checking the state of the ISA (Instruction Set Architecture) field in the *Config3* CP0 register. *Config1*<sub>CA</sub> (bit 2) is not used for microMIPS.

Software can determine if the MIPS32 ISA is implemented by checking the state of the ISA (Instruction Set Architecture) register field in the *Config3* CP0 register.

Software can determine which ISA is used when handling an exception by checking the state of the ISAOnExc (ISA on Exception) field in the *Config3* CP0 register.

Debug Probe Software can determine which ISA is used when handling a debug exception by checking the state of the ISAOnDebug field in the *EJTAG TAP Control* register.

### 11.1.4 Compliance and Subsetting

This document does not change the instruction subsets as defined by the other MIPS32 architecture reference manuals, including the subsets defined by the various ASEs.

### 11.1.5 Mode Switch

The MIPS32 architecture defines an ISA mode for each processor. An ISA mode value of 0 indicates MIPS32 instruction decoding. In processors implementing microMIPS, an ISA mode value of 1 selects microMIPS instruction decoding. In processors implementing the MIPS16e ASE, an ISA mode value of 1 selects the decoding of instructions as MIPS16e.

In microMIPS implementations, the ISA mode is not directly visible to normal software. When EJTAG is implemented, the ISAMode is reflected in the *EJTAG TAP Control* register.

Mode switching between MIPS32 and microMIPS uses the same mechanism used by MIPS16e, namely, the JALX, JR, JR.HB, JALR, and JALR.HB instructions, as described below.

- The JALX instruction executes a JAL and switches to the other mode.
- The JR and JALR instructions interpret bit 0 of the source registers as the target ISA mode (0=MIPS32, 1=microMIPS) and therefore set the ISA Mode bit according to the contents of bit 0 of the source register. For the actual jump operation, the PC is loaded with the value of the source register with bit 0 set to 0. The same applies to JR.HB and JALR.HB. The instructions JALR and JALR.HB save the ISA mode into bit 0 of the destination register.
- When exceptions or interrupts occur and the processor writes to *EPC*, *DEPC*, or *ErrorEPC*, the ISA Mode bit is saved into bit 0 of these registers. Then the ISA Mode bit is set according to the *Config3*<sub>ISA</sub> register field. On return from an exception, the processor loads the ISA Mode bit based on the value from either *EPC*, *DEPC*, or *ErrorEPC*.

If only one ISA mode exists (either MIPS32 or microMIPS) then this mode switch mechanism does not exist, and the ISA mode has a fixed value (0=MIPS32, 1=microMIPS). Executing the JALX instruction will cause a Reserved Instruction exception. JR and JALR instructions cause an Address exception on the target instruction fetch when bit 0 of the source register is different from the ISA mode. The same applies to JR.HB and JALR.HB. Exception handlers must be encoded in the instruction format supported by the processor.

### 11.1.6 Branch and Jump Offsets

In the MIPS32 architecture, because instructions are always 32 bits in size, the jump and branch target addresses are word (32-bit) aligned. Jump/branch offset fields are shifted left by two bits to create a word-aligned effective address.

In the microMIPS architecture, because instructions can be either 16 or 32 bits in size, the jump and branch target addresses are halfword (16-bit) aligned. Branch/jump offset fields are shifted left by only one bit to create halfword-aligned effective addresses.

To maintain the existing MIPS32 ABIs, link unit/object file entry points are restricted to 32-bit word alignments. In the future, a microMIPS-only ABI can be created to remove this restriction.

### 11.1.7 Coprocessor Unusable Behavior

If an instruction associated with a non-implemented coprocessor is executed, it is implementation-specific whether a processor executing in microMIPS mode raises an RI exception or a coprocessor unusable exception. While in microMIPS mode, the M14K has the same behavior as in MIPS32 mode; coprocessor unusable exceptions will be raised.

## 11.2 Instruction Formats

This section defines the formats of microMIPS instructions. The microMIPS variable-length encoding comprises 16-bit and 32-bit wide instructions.

The 6-bit major opcode is left-aligned within the instruction encoding. Instructions can have 0 to 4 register fields. For 32-bit instructions, the register field width is 5 bits, while for most 16-bit instructions, the register field width is 3 bits, utilizing instruction-specific register encoding. All 5-bit register fields are located at a constant position within the instruction encoding.

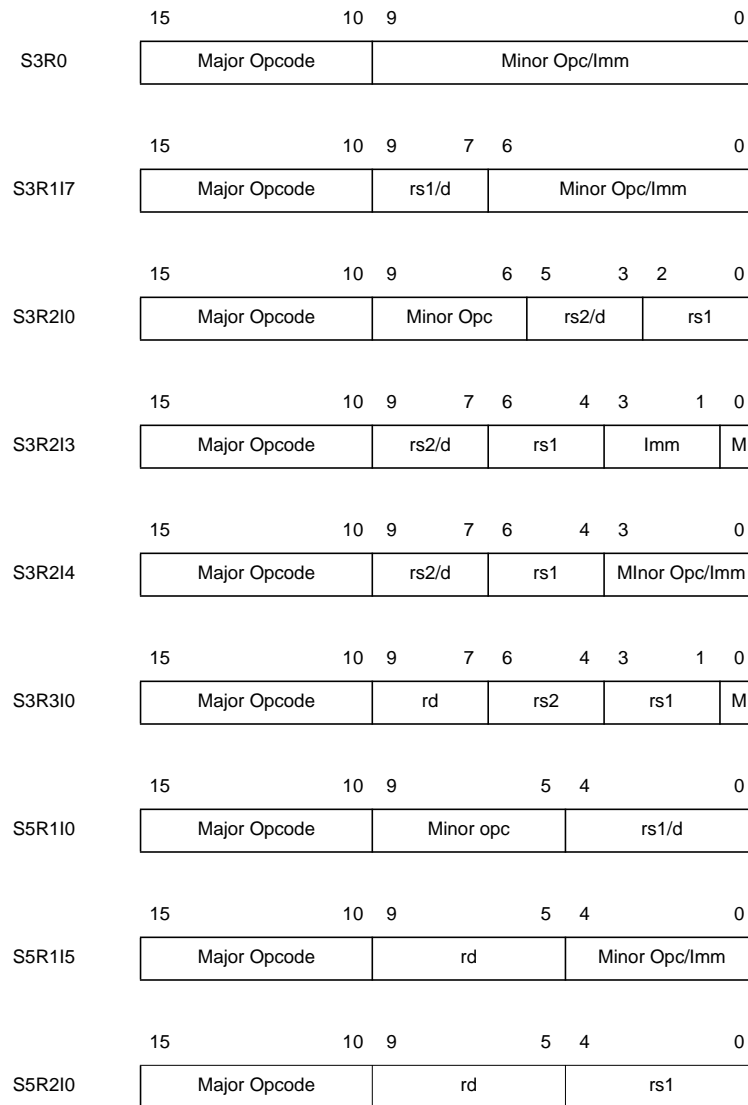
The immediate field is right-aligned in the following instructions:

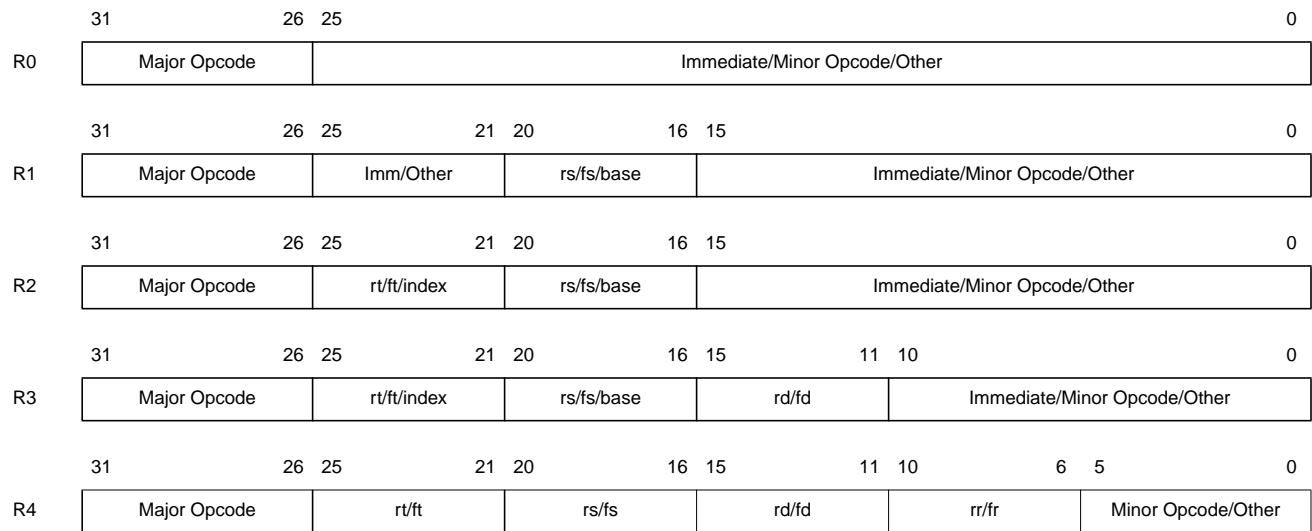
- some 16-bit instructions with 3-bit register fields
- 32-bit instructions with 16-bit or 26-bit immediate field

The name ‘immediate field’ as used here includes the address offset field for branches and load/store instructions as well as the jump target field.

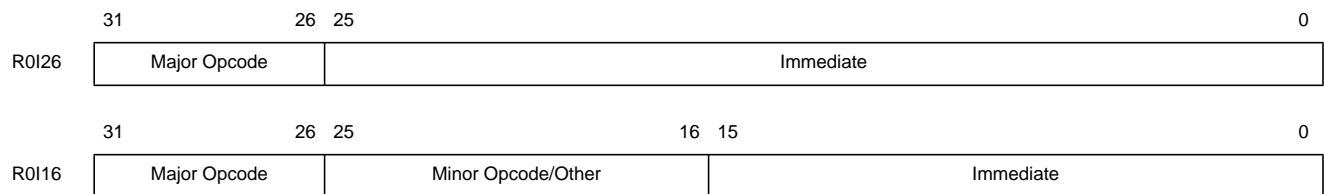
Other instruction-specific fields are typically located between the immediate and minor opcode fields. Instructions that have multiple “other” fields are listed in alphabetical order according to the name of the field, with the first name of the order located at the lower bit position. An empty bit field that is not explicitly shown in the instruction format is located next to the minor opcode field.

**I** [Figure 11.1](#) and [Figure 11.2](#) show the 16-bit and 32-bit instruction formats.

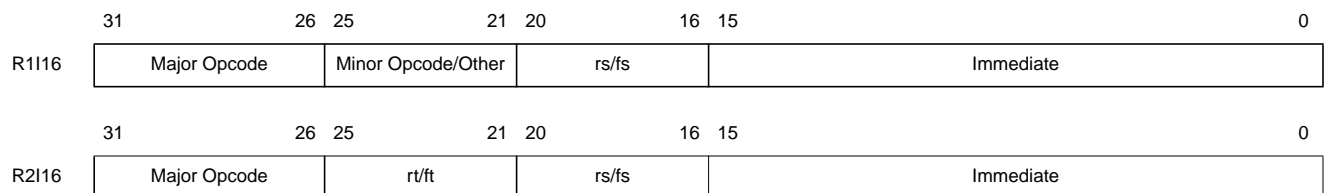
**Figure 11.1 16-Bit Instruction Formats**

**Figure 11.2 32-Bit Instruction Formats****Figure 11.3 Immediate Fields within 32-Bit Instructions**

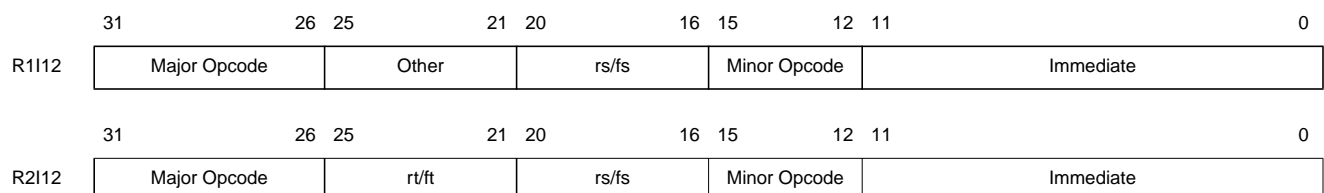
32-bit instruction formats with 26-bit immediate fields:



32-bit instruction formats with 16-bit immediate fields:



32-bit instruction formats with 12-bit immediate fields:





The instruction size can be completely derived from the major opcode. For 32-bit instructions, the major opcode also defines the position of the minor opcode field and whether or not the immediate field is right-aligned.

Instructions formats are named according to the number of the register fields and the size of the immediate field. The names have the structure R<x>I<y>. For example, an instruction based on the format R2I16 has 2 register fields and a 16-bit immediate field.

### 11.2.1 Instruction Stream Organization and Endianness

16-bit instructions are placed within the 32-bit (or 64-bit) memory element according to system endianness.

- On a 32-bit processor in big-endian mode, the first instruction is read from bits 31..16, and the second instruction is read from bits 15..0.
- On a 32-bit processor in little-endian mode, the first instruction is read from bits 15..0, and the second instruction is read from bits 31..16.

The above rule also applies to the halfwords of 32-bit instructions. This means that a 32-bit instruction is not treated as a word data type; instead, the halfwords are treated in the same way as individual 16-bit instructions. The halfword containing the major opcode is always the first in the sequence.

Example:

SRL r1, r1, 7	binary opcode fields:	000000 00001 00001 00111 00001 000000
	hex representation:	0021 3840
	Address:	3 2 1 0
Little Endian:	Data:	38 40 00 21
	Address:	0 1 2 3
Big Endian:	Data:	00 21 38 40

Instructions are placed in memory such that they are in-order with respect to the address.

## 11.3 microMIPS Re-encoded Instructions

This section lists all microMIPS re-encoded instructions, sorted into 16-bit and 32-bit categories.

In the 16-bit category:

- Frequent MIPS32 instructions and macros, re-encoded as 16-bit. Register and immediate fields are reduced in size by using encodings of frequently occurring values.

In the 32-bit category:

- All MIPS32 instructions, including all application-specific extensions except MIPS16e, re-encoded: MIPS32, MIPS-3D ASE, MIPS DSP ASE, MIPS MT ASE, and SmartMIPS ASE.
- Opcode space for user-defined instructions (UDIs).
- New instructions designed primarily to reduce code size.

To differentiate between 16-bit and 32-bit encoded instructions, the instruction mnemonic can be optionally extended with the suffix “16” or “32” respectively. This suffix is placed at the end of the instruction before the first ‘.’ if there is one. For example:

ADD16, ADD32, ADD32.PS

If these suffixes are omitted, the assembler automatically chooses the smallest instruction size.

For each instruction, the tables in this chapter provide all necessary information about the bit fields. The formats of the instructions are defined in [Section 9.1 “CPU Instruction Formats”](#). Together with the major and minor opcode encodings, which can be derived from the tables in [Section 10.2 “M14K™ Opcode Map”](#), the complete instruction encoding is provided.

Most register fields have a width of 5 bits. 5-bit register fields use linear encoding ( $r0=’00000’$ ,  $r1=’00001’$ , etc.). For 16-bit instructions, whose register field size is variable, the register field width is explicitly stated in the instruction table ([Table 11.1](#) and [Table 11.2](#)), and the individual register and immediate encodings are shown in [Table 11.3](#). The ‘other fields’ are defined by the respective column, with the order of these fields in the instruction encoding defined by the order in the tables.

## 11.3.1 16-Bit Category

### 11.3.1.1 Frequent MIPS32 Instructions

These are frequent MIPS32 instructions with reduced register and immediate fields containing frequently used registers and immediate values.

MOVE is a very frequent instruction. It therefore supports full 5-bit unrestricted register fields for maximum efficiency. In fact, MOVE used to be a simplified macro of an existing MIPS32 instruction.

There are 2 variants of the LW and SW instructions. One variant implicitly uses the SP register to allow for a larger offset field. The value in the offset field is shifted left by 2 before it is added to the base address.

There are four variants of the ADDIU instruction:

1. A variant with one 5-bit register specifier that allows any GPR to be the source and destination register
2. A variant that uses the stack pointer as the implicit source and destination register
3. A variant that has separate 3-bit source and destination register specifiers
4. A variant that has the stack pointer as the implicit source register and one 3-bit destination register specifier

A 16-bit NOP instruction is needed because of the new 16-bit instruction alignment and the need in specific cases to align instructions on a 32-bit boundary. It can save code size as well. NOP is not shown in the table because it is realized as a macro (as is NEGU).

```
NOP16 = MOVE16 r0, r0
```

```
NEGU16 rt, rs = SUBU16 rt, r0, rs
```

Because microMIPS instructions are 16-bit aligned, the 16-bit branch instructions support 16-bit aligned branch target addresses. The offset field is left shifted by 1 before it is added to the PC.

The compact instruction JRC is to be used instead of JR, when the jump delay slot after JR cannot be filled. This saves code size. Because JRC may execute as fast as JR with a NOP in the delay slot, JR is preferred if the delay slot can be filled.

The breakpoint instructions, BREAK and SDBBP, include a 16-bit variant that allows a breakpoint to be inserted at any instruction address without overwriting more than a single instruction.

**Table 11.1 16-Bit Re-encoding of Frequent MIPS32 Instructions**

Instruction	Major Opcode Name	Number of Register Fields	Immediate Field Size (bit)	Register Field Width (bit)	Total Size of Other Fields	Empty 0 Field Size (bit)	Minor Opcode Size (bit)	Comment
ADDIU5	POOL16D	5bit:1	4	5		0	1	Add Immediate Unsigned Word Same Register
ADDIUSP	POOL16D	0	9	0		0	1	Add Immediate Unsigned Word to Stack Pointer
ADDIUR2	POOL16E	2	3	3		0	1	Add Immediate Unsigned Word Two Registers
ADDIUR1SP	POOL16E	1	6	3		0	1	Add Immediate Unsigned Word One Registers and Stack Pointer
ADDU16	POOL16A	3	0	3		0	1	Add Unsigned Word
AND16	POOL16C	2	0	3		0	4	AND
ANDI16	ANDI16	2	4	3		0	0	AND Immediate
B16	B16	0	10			0	0	Branch
BREAK16	POOL16C	0	0		4	0	6	Cause Breakpoint Exception
JALR16	POOL16C	1	0	5		0	5	Jump and Link Register, 32-bit delay-slot
JALRS16	POOL16C	1	0	5		0	5	Jump and Link Register, 16-bit delay-slot
JR16	POOL16C	1	0	5		0	5	Jump Register
LBU16	LBU16	2	4	3		0	0	Load Byte Unsigned
LHU16	LHU16	2	4	3		0	0	Load Halfword
LI16	LI16	1	7	3		0	0	Load Immediate
LW16	LW16	2	4	3		0	0	Load Word
LWGP	LWGP16	1	7	3		0	0	Load Word GP
LWSP	LWSP16	5bit:1	5	5		0	0	Load Word SP

**Table 11.1 16-Bit Re-encoding of Frequent MIPS32 Instructions (Continued)**

Instruction	Major Opcode Name	Number of Register Fields	Immediate Field Size (bit)	Register Field Width (bit)	Total Size of Other Fields	Empty 0 Field Size (bit)	Minor Opcode Size (bit)	Comment
MFHI16	POOL16C	1	0	5		0	5	Move from HI Register
MFLO16	POOL16C	1	0	5		0	5	Move from LO Register
MOVE16	MOVE16	2	0	5		0	0	Move
NOT16	POOL16C	2	0	3		0	4	NOT
OR16	POOL16C	2	0	3		0	4	OR
SB16	SB16	2	4	3		0	0	Store Byte
SDBBP16	POOL16C	0	0		4	0	6	Cause Debug Breakpoint Exception
SH16	SH16	2	4	3		0	0	Store Halfword
SLL16	POOL16B	2	3	3		0	1	Shift Word Left Logical
SRL16	POOL16B	2	3	3		0	1	Shift Word Right Logical
SUBU16	POOL16A	3	0	3		0	1	Sub Unsigned
SW16	SW16	2	4	3		0	0	Store Word
SWSP	SWSP16	5bit:1	5	5		0	0	Store Word SP
XOR16	POOL16C	2	0	3		0	4	XOR

**11.3.1.2 Frequent MIPS32 Instruction Sequences**

These 16-bit instructions are equivalent to frequently-used short sequences of MIPS32 instructions. The instruction-specific register and immediate value selection are shown in [Table 11.3](#).

**Table 11.2 16-Bit Re-encoding of Frequent MIPS32 Instruction Sequences**

Instruction	Major Opcode Name	Number of Register Fields	Immediate Field Size (bit)	Register Field Width (bit)	Total Size of Other Fields	Empty 0 Field Size (bit)	Minor Opcode Size (bit)	Comment
BEQZ16	BEQZ16	1	7	3		0	0	Branch on Equal Zero
BNEZ16	BNEZ16	1	7	3		0	0	Branch on Not Equal Zero
JRADDIUSP	POOL16C	0	5				5	Jump Register; ADDIU SP
JRC	POOL16C	1	0	5		0	5	Jump Register Compact

**Table 11.2 16-Bit Re-encoding of Frequent MIPS32 Instruction Sequences (Continued)**

<b>Instruction</b>	<b>Major Opcode Name</b>	<b>Number of Register Fields</b>	<b>Immediate Field Size (bit)</b>	<b>Register Field Width (bit)</b>	<b>Total Size of Other Fields</b>	<b>Empty 0 Field Size (bit)</b>	<b>Minor Opcode Size (bit)</b>	<b>Comment</b>
LWM16	POOL16C	0	4		2	0	4	Load Word Multiple
SWM16	POOL16C	0	4		2	0	4	Store Word Multiple

## 11.3.1.3 Instruction-Specific Register Specifiers and Immediate Field Encodings

Table 11.3 Instruction-Specific Register Specifiers and Immediate Field Values

Instruction	Number of Register Fields	Immediate Field Size (bit)	Register 1 Decoded Value	Register 2 Decoded Value	Register 3 Decoded Value	Immediate Field Decoded Value
ADDIU5	5bit:1	4	rd: 5-bit field			-8..0..7
ADDIUSP	0	9				$(-258..-3, 2..257) \ll 2$
ADDIUR2	2	3	rs1:2-7,16, 17	rd:2-7,16, 17		-1, 1, 4, 8, 12, 16, 20, 24
ADDIUR1SP	1	6	rd:2-7,16, 17			$(0..63) \ll 2$
ADDU16	3	0	rs1:2-7,16, 17	rs2:2-7,16, 17	rd:2-7,16, 17	
AND16	2	0	rs1:2-7,16, 17	rd:2-7,16, 17		
ANDI16	2	4	rs1:2-7,16, 17	rd:2-7,16, 17		1, 2, 3, 4, 7, 8, 15, 16, 31, 32, 63, 64, 128, 255, 32768, 65535
B16	0	10				$(-512..511) \ll 1$
BEQZ16	1	7	rs1:2-7,16, 17			$(-64..63) \ll 1$
BNEZ16	1	7	rs1:2-7,16, 17			$(-64..63) \ll 1$
BREAK16	0	4				0..15
JALR16	5bit:1	0	rs1:5-bit field			
JALRS16	5bit:1	0	rs1:5-bit field			
JRADDIUSP	0	5				$(0..31) \ll 2$
JR16	5bit:1	0	rs1:5 bit field			
JRC	5bit:1	0	rs1:5 bit field			
LBU16	2	4	rb:2-7,16,17	rd:2-7,16, 17		-1,0..14
LHU16	2	4	rb:2-7,16,17	rd:2-7,16, 17		$(0..15) \ll 1$
LI16	1	7	rd:2-7,16, 17			-1,0..126
LW16	2	4	rb:2-7,16,17	rd:2-7,16, 17		$(0..15) \ll 2$
LWM16	2bit list:1	4				$(0..15) \ll 2$
LWGP	1	7	rd:2-7,16,17			$(-64..63) \ll 2$
LWSP	5bit:1	5	rd:5-bit field			$(0..31) \ll 2$
MFHI16	5bit:1	0	rd:5-bit field			
MFLO16	5bit:1	0	rd:5-bit field			
MOVE16	5bit:2	0	rd:5-bit field	rs1:5-bit field		
NOT16	2	0	rs1:2-7,16, 17	rd:2-7,16, 17		
OR16	2	0	rs1:2-7,16, 17	rd:2-7,16, 17		
SB16	2	4	rb:2-7,16,17	rs1:0, 2-7, 17		0..15
SDBBP16	0	0				0..15
SH16	2	4	rb:2-7,16,17	rs1:0, 2-7, 17		$(0..15) \ll 1$
SLL16	2	3	rs1:2-7,16, 17	rd:2-7,16, 17		1..8 (see encoding tables)

**Table 11.3 Instruction-Specific Register Specifiers and Immediate Field Values (Continued)**

Instruction	Number of Register Fields	Immediate Field Size (bit)	Register 1 Decoded Value	Register 2 Decoded Value	Register 3 Decoded Value	Immediate Field Decoded Value
SRL16	2	3	rs1:2-7,16, 17	rd:2-7,16, 17		1..8 (see encoding tables)
SUBU16	3	0	rs1:2-7,16, 17	rs2:2-7,16, 17	rd:2-7,16, 17	
SW16	2	4	rb:2-7,16,17	rs1:0, 2-7, 17		(0..15) << 2
SWSP	5bit:1	5	rs1: 5 bit field			(0..31) << 2
SWM16	2- bit list:1	4				(0..15)<<2
XOR16	2	0	rs1:2-7,16, 17	rd:2-7,16, 17		

### 11.3.2 16-bit Instruction Register Set

Many of the 16-bit instructions use 3-bit register specifiers in their binary encodings. The register set used for most of these 3-bit register specifiers is listed in [Table 11.4](#). The register set used for SB16, SH16, SW16 source register is listed in [Table 11.5](#). These register sets are a true subset of the register set available in 32-bit mode; the 3-bit register specifiers can directly access 8 of the 32 registers available in 32-bit mode (which uses 5-bit register specifiers).

In addition, specific instructions in the 16-bit instruction set implicitly reference the stack pointer register (*sp*), global pointer register (*gp*), the return address register (*ra*), the integer multiplier/divider output registers (*HI/LO*) and the program counter (*PC*). Of these, [Table 11.6](#) lists *sp*, *gp* and *ra*. [Table 11.7](#) lists the microMIPS special-purpose registers, including *PC*, *HI* and *LO*.

The microMIPS also contains some 16-bit instructions that use 5-bit register specifiers. Such 16-bit instructions provide access to all 32 general-purpose registers.

**Table 11.4 16-Bit Instruction General-Purpose Registers - \$2-\$7, \$16, \$17**

16-Bit Register Encoding <sup>1</sup>	32-Bit MIPS Register Encoding <sup>2</sup>	Symbolic Name (From <i>ArchDefs.h</i> )	Description
0	16	s0	General-purpose register
1	17	s1	General-purpose register
2	2	v0	General-purpose register
3	3	v1	General-purpose register
4	4	a0	General-purpose register
5	5	a1	General-purpose register
6	6	a2	General-purpose register
7	7	a3	General-purpose register

1. “0-7” correspond to the register’s 16-bit binary encoding and show how that encoding relates to the MIPS registers. “0-7” never refer to the registers, except within the binary microMIPS instructions. From the assembler, only the MIPS names (\$16, \$17, \$2, etc.) or the symbolic names (s0, s1, v0, etc.) refer to the registers. For example, to access register number 17 in the register file, the programmer references \$17 or s1, even though the microMIPS binary encoding for this register is 001.
2. General registers not shown in the above table are not accessible through the 16-bit instruction using 3-bit register specifiers. The Move instruction can access all 32 general-purpose registers.

**Table 11.5 SB16, SH16, SW16 Source Registers - \$0, \$2-\$7, \$17**

16-Bit Register Encoding <sup>1</sup>	32-Bit MIPS Register Encoding <sup>2</sup>	Symbolic Name (From <i>ArchDefs.h</i> )	Description
0	0	zero	Hard-wired Zero
1	17	s1	General-purpose register
2	2	v0	General-purpose register
3	3	v1	General-purpose register
4	4	a0	General-purpose register
5	5	a1	General-purpose register
6	6	a2	General-purpose register
7	7	a3	General-purpose register

1. “0-7” correspond to the register’s 16-bit binary encoding and show how that encoding relates to the MIPS registers. “0-7” never refer to the registers, except within the binary microMIPS instructions. From the assembler, only the MIPS names (\$16, \$17, \$2, etc.) or the symbolic names (s0, s1, v0, etc.) refer to the registers. For example, to access register number 17 in the register file, the programmer references \$17 or s1, even though the microMIPS binary encoding for this register is 001.
2. General registers not shown in the above table are not accessible through the 16-bit instructions using 3-bit register specifier. The Move instruction can access all 32 general-purpose registers.

**Table 11.6 16-Bit Instruction Implicit General-Purpose Registers**

16-Bit Register Encoding	32-Bit MIPS Register Encoding	Symbolic Name (From <i>ArchDefs.h</i> )	Description
Implicit	28	gp	Global pointer register
Implicit	29	sp	Stack pointer register
Implicit	31	ra	Return address register



**Table 11.7 16-Bit Instruction Special-Purpose Registers**

Symbolic Name	Purpose
PC	Program counter. The PC-relative ADDIU can access this register as an operand.
HI	Contains high-order word of multiply or divide result.
LO	Contains low-order word of multiply or divide result.

### 11.3.3 32-Bit Category

#### 11.3.3.1 New 32-bit instructions

The following table lists the 32-bit instructions introduced in the microMIPS ISA.

**Table 11.8 32-bit Instructions introduced within microMIPS**

Instruction	Major Opcode Name	Number of Register Fields	Immediate Field Size (bit)	Register Field Width (bit)	Total Size of Other Fields	Empty 0 Field Size (bit)	Minor Opcode Size (bit)	Comment
ADDIUPC	ADDIUPC	1	23	3		0	0	ADDIU PC-Relative
BEQZC	POOL32I	2:5 bit	16	5			0	Branch on Equal to Zero, No Delay Slot
BNEZC	POOL32I	2:5 bit	16	5			0	Branch on Not Equal to Zero, No Delay Slot
JALRS	POOL32A	2:5 bit	0	5			16	Jump and Link Register, Short Delay Slot
JALRS.HB	POOL32A	2:5 bit	0	5			16	Jump and Link Register with Hazard Barrier, Short Delay Slot
JALS	JALS32	0	26				0	Jump and Link, Short Delay Slot
JALX	JALX		26	5		0	5	Jump and Link Exchange
LWP	POOL32B	2:5 bit	12		5	0	4	Load Word Pair
LWXS	POOL32A	3:5 bit	0	5	0	1	10	Load Word Indexed, Scale
LWM32	POOL32B	1:5bit	12		5	0	4	Load Word Multiple
SWP	POOL32B	2:5 bit	12			0	4	Load Word Pair
SWM32	POOL32B	1:5bits	12		5	0	4	Store Word Multiple



## References

This appendix lists other publications available from MIPS Technologies, Inc. that are referenced in this document. These documents may be included in the `$MIPS_PROJECT/doc` area of a typical M14K soft or hard core release, or in some cases may be available on the MIPS web site <http://www.mips.com>.

1. MIPS32® M14K™ Processor Core Data Sheet  
MIPS document: MD00666
2. MIPS32® M14K™ Processor Core Integrators Guide  
MIPS document: MD00667
3. MIPS32® M14K™ CPU Implementor's Guide  
MIPS Document: MD00669
4. MIPS32® M14K™ System Package & Simulation Flow User's Manual  
MIPS document: MD00717
5. MIPS® Architecture For Programmers, Volume I: Introduction to the MIPS32® Architecture  
MIPS document: MD0082
6. MIPS® Architecture For Programmers, Volume I: Introduction to the microMIPS32™ Architecture  
MIPS document: MD0741
7. MIPS® Architecture For Programmers, Volume II: The MIPS32® Instruction Set  
MIPS document: MD0086
8. MIPS® Architecture For Programmers, Volume II: The microMIPS32™ Instruction Set  
MIPS document: MD0582
9. MIPS® Architecture For Programmers Volume III: The MIPS32® and microMIPS32™ Privileged Resource Architecture  
MIPS Document: MD00090
10. MIPS® Architecture for Programmers Volume IV-h: The MCU Application-Specific Extension to the MIPS32® and microMIPS32™ Architectures  
MIPS document: MD00641
11. MIPS® EJTAG Specification  
MIPS Document: MD00047
12. MIPS® cJTAG Adapter User's Manual  
MIPS Document: MD00862
13. Security Features of the M14K™ Processor Family  
MIPS document: MD00896

## References

## Revision History

Change bars (vertical lines) in the margins of this document indicate significant changes in the document since its last release. Change bars are removed for changes that are more than one revision old.

This document may refer to Architecture specifications (for example, instruction set descriptions and EJTAG register definitions), and change bars in these sections indicate changes since the previous version of the relevant Architecture document.

Revision	Date	Description
01.00	March 25, 2010	• Initial 1_0_0 release.
02.00	December 17, 2010	• 2_0_0 Maintenance release.
02.01	September 30, 2011	• 2_1_0 Maintenance release.
02.02	March 12, 2012	• 2_1a_0 Patch release.
02.03	April 30, 2012	• 2_2_0 Maintenance release.

## Revision History